GC-Consistent Cuts of Databases

Marcin Skubiszewski, Nicolas Porteix

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

GC-consistent Cuts of Databases

Marcin Skubiszewski and Nicolas Porteix

N° 2681

Octobre 1995

PROGRAMME 1

ISSN 0249-6399

GC-consistent Cuts of Databases

Marcin Skubiszewski and Nicolas Porteix

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet RODIN, work done in collaboration with O_2 Technology

Rapport de recherche n°2681 — Octobre 1995 — 50 pages

Abstract: We introduce a new method for concurrent mark-and-sweep garbage collection in object-oriented databases. For this purpose, we define a *cut* of a database to be a collection containing one or more copies of every page in the database; the copies may have been made at different times during the operation of the database. We define a class of cuts called *GC-consistent cuts*, and prove formally that a garbage collector can correctly determine which objects to delete by examining a GC-consistent cut of a database instead of the database itself. We show that GC-consistent cuts can synchronize the concurrent collector with the mutator, *i.e.* perform the task usually assigned to a write barrier: while a database is in operation, a GC-consistent cut of it can be built in an efficient and inobtrusive way, and, while still under construction, can be used by a garbage collector.

We investigate other fundamental properties of GC-consistent cuts. We compare their consistency properties with those of causal cuts of distributed systems. We show that although the reachability of objects in a GC-consistent cut is inherited from the underlying database, many other interesting properties of the cut are unrelated to those of the database; this weak consistency is related to the low cost of building GC-consistent cuts.

Key-words: concurrent garbage collection, object-oriented database, causal cut, GC-consistent cut, write barrier, distributed system.

(Résumé : tsvp)

Work partly supported by O_2 Technology (7 rue du Parc de Clagny, 78000 Versailles, France) and by Ecole Polytechnique (91128 Palaiseau, France).
E-mail: `Marcin.Skubiszewski@inria.fr` and `Nicolas.Porteix@enst.fr`

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Coupes de bases de données cohérentes pour les ramasse-miettes

Résumé : Nous introduisons une nouvelle méthode de ramassage concurrent de miettes dans des bases de données à objets. Pour cela, nous introduisons la notion de *coupe* d'une base de données, qui est une collection contenant une ou plusieurs copies de chaque page de la base ; les copies peuvent avoir été faites à des instants différents durant le fonctionnement de la base. Nous introduisons les *coupes cohérentes pour les ramasse-miettes*, ou *coupes GC-cohérentes*, et nous démontrons qu'un ramasse-miettes peut déterminer correctement quels objets d'une base de données doivent être détruits en examinant une coupe GC-cohérente de la base à la place de la base elle-même. Nous expliquons comment les coupes GC-cohérentes peuvent servir à synchroniser le ramasse-miettes concurrent avec le mutateur, c'est-à-dire accomplir la tâche habituellement assignée aux barrières d'écriture. Pendant qu'une base de données fonctionne, une coupe GC-cohérente de cette base peut être construite efficacement et, durant sa construction, être utilisée par un ramasse-miettes.

Nous décrivons d'autres propriétés fondamentales des coupes GC-cohérentes. Nous comparons leur propriétés de cohérence avec celles des coupes causales de systèmes répartis. Nous montrons que dans une coupe GC-cohérente seule l'atteignabilité des objets est héritée de la base de données correspondante, tandis que les autres propriétés sont, pour la plupart, indépendantes des propriétés de la base ; cette cohérence faible est liée au fait que les coupes GC-cohérentes peuvent être construites à faible coût.

Mots-clé : ramassage de miettes concurrent, base de données à objets, coupe causale, coupe GC-cohérente, barrière d'écriture, système réparti.

1 Introduction

Automatic garbage collection is widely recognized as a fundamental mechanism which needs to be provided to the developpers of application software. Garbage collection is usually provided by system software, *i.e.* by the operating system, by the database management system (DBMS) or by the runtime environment of a programming language. Unfortunately, garbage collectors (GCs) tend to be highly obtrusive: their operation is resource-consuming and imposes inconvenient synchronization requirements upon the rest of the system. This report addresses the question of garbage collection in databases. It describes a new method for solving the synchronization problems which appear in this context.

In this work, we assume that the garbage collector does not modify the system in a way observable by the mutator (the term *mutator*, introduced by Dijkstra *et al.* [9] and now widely accepted, represents all the activities in the system except the GC). This assumption is in agreement with the purpose of garbage collection, which is to delete *garbage* objects, that is, objects to which the mutator does not have access; other objects are not supposed to be modified or moved by a GC, at least in principle. The assumption is actually satisfied by *marking and sweeping*, a garbage collection mechanism well suited for database systems; a description of marking and sweeping, and a concise comparison between this and other garbage collection mechanisms, are given later on (Section 2.5).

Under our assumption, there is no need to protect the mutator from the modifications introduced into the system by the collector. But the converse is not true. In order to decide which objects are garbage, the GC needs to examine the system in its entirety, and if the system is arbitrarily modified during this examination, incorrect decisions may result. The GC must impose synchronization requirements sufficiently strong to suppress this problem. With unsophisticated garbage collectors, these requirements are extreme: for the time of a complete execution of the GC, the mutator is blocked. In some systems, *e.g.* in huge databases, an execution of the GC may last up to several hours. It is annoying, if at all acceptable, to have the system blocked for such an amount of time.

Unsophisticated GCs block all other activities because they are built upon the simplistic idea that in order to view the system consistently, we must prevent it from being modified while we examine it. The modern, more sophisticated GCs can execute concurrently with the mutator. This implies that the system may be modified at any time while the GC is analyzing it. The classical principle for maintaining correctness despite of this is known as *write barriers*.[1] According to this principle, the GC makes no effort to obtain a consistent view of the system: each object is seen in the state in which it happens to be when the GC looks at it. Instead, the mutator notifies the GC of every pointer modification performed while the GC is running. For this purpose, the mutator code is instrumented, or virtual memory mechanisms are used to detect writes, or, in systems with logging, the log is made available to the GC and analyzed by it.

[1] The first algorithm based on write barriers was introduced by Dijkstra *et al.* [9]. A comprehensive survey of concurrent garbage collection techniques by Wilson [20] includes a detailed discussion of write barriers. Yong *et al.* [21] measure the performance of several classical garbage collection schemes in the context of a persistent store. Hosking *et al.* [14] measure the performance of several implementation of write barriers. Concurrent GCs which use write barriers to collect persistent objects include [3, 18, 2, 15]. See also a barrier-based GC by Doligez and Gonthier [10].

The notifications are used by the GC to build a list of objects which were reachable at some point during the garbage detection process, yet which might be improperly seen as unreachable. The GC considers all objects in the list as reachable; this is sufficient to ensure correctness.

Unfortunately, this form of mutator-collector communication is undesirable in many systems. Let us mention here its two most important drawbacks. First, the implementations where mutator code is instrumented are incompatible with existing executable code. In many systems, even user code needs to be compiled in a special, GC-compatible way. Second, the mutator-collector communication seriously increases the cost of every pointer update performed by the mutator. In fact, the communication is more resource-consuming than the update itself, which amounts to an ordinary memory write. Some overhead remains even while the GC is not running (and thus no actual communication occurs). The latter fact is obvious in the implementations where user code is instrumented: whenever a pointer is modified, the instrumented code needs at least to check whether a GC is running. It is nonobvious, but still real, in the other implementations [3, 18].

These drawbacks lead us to consider a new principle for concurrent garbage collection. We build a static view of the system which *contains no false garbage*: an object is seen as garbage in the view only if it remains constantly garbage in the real system. The GC examines the view instead of examining the real, ever-changing system. This eliminates the need for the GC to receive corrective information, and thus allows us to get rid of mutator-collector communication, which is expensive and complicated.

The key problem lies in building such a static view while the mutator is running. This report solves the problem in the case of databases. We introduce a category of views of databases called *GC-consistent cuts*. GC-consistent cuts contain no false garbage, and can be built concurrently, while the mutator is running.

GC-consistent cuts are a way to view a database consistently, yet without observing all parts of it at the same time. They are therefore similar to *causal cuts*,[2] which serve the same purpose in the world of asynchronous distributed systems. There is, however, an important difference between these concepts. Causal cuts are intended to represent *the* correct way to observe an asynchronous distributed system, for all reasonable purposes. GC-consistent cuts, on the other hand, only represent the proper way to observe a database for the purposes of garbage collection. For other purposes, other kinds of cuts are expected to be used. In this report, besides the GC-consistent cuts, we introduce two other kinds of cuts, called respectively *GC-consistent simple cuts* and *causal cuts of databases*; we compare the properties of the three categories of cuts.

The report is organized as follows. In Section 2, we recall basic facts about databases, introduce the key concepts used in the report, and state the consistency properties which a cut must satisfy in order to be useful for garbage collection. In Section 3, we define GC-consistent cuts and we state that they satisfy the previously-defined consistency properties; the proof of this statement is deferred to Appendix A. In Section 4, we explain how GC-consistent cuts can be used in practice; among others, we explain how they can be built cheaply and concurrently. Section 5 deals with theoretical aspects of GC-consistent cuts, and introduces the two other above-mentionned kinds of cuts. It compares the

[2] Chandy and Lamport [8] introduced a "global-state recording algorithm" which implements one kind of causal cut; then, Mattern [17] defined causal cuts in the general case. Babaoglu and Marzullo [5] describe causal cuts in detail.

properties of GC-consistent cuts with those of causal cuts. Section 6 summarizes our contribution. Appendix A contains proofs.

2 Fundamental definitions and assumptions

In this section we recall some concepts related to databases and to garbage collection. We introduce a graphical notation for describing executions of databases; this notation is similar to the one generally used for describing causality in distributed systems. We define cuts.

2.1 Transactions and their properties

An execution (that is, a sequence of operations performed on a database) is divided into chunks called *transactions*. Each transaction locks the data to which it has access. A lock permits a transaction either only to read or both to read and to write specified data. By monitoring locks, an observer can know which data are read or modified by any given transaction. This knowledge is an essential prerequisite for building GC-consistent cuts of a database.

We assume *atomicity*: every transaction either commits or aborts. The effects of a committed transaction are fully taken into account. An aborted transactions has no effect; if some effects of it have been taken into account by the system before the abort, these effects are fully undone at abort time. In our model, only comitted transactions are taken into account; aborted transactions are entirely disregarded.

We assume that transactions are *serialisable*, *i.e.* that everything happens as if they were executed sequentially, in some specified order. In reality, transactions may be executed concurrently, and serialisability is implemented by the locking mechanism, which permits concurrent execution only when it is indistinguishable from a sequential one. Serialisability, and the resulting apparent lack of concurrency, allows us to depict each transaction as an atomic (thus, null-duration) event, which takes place at the time when the actual transaction commits.

2.2 Parts and addresses

We view data in a database as being partitioned into parts numbered $0, ..., n-1$. Each object x belongs to exactly one part, noted $P(x)$. We assume that by looking at the address of x (*i.e.* at the value of a pointer to x), one can determine to which part x belongs. Usually, parts are database pages, but alternatively their rôle can be played by objects (in this case, $P(x) = x$) or by other entities.

In our model, objects do not migrate: for every x, $P(x)$ remains constant over time. Similarly, the address of x, *i.e.* the physical representation of pointers to x, does not change over time. This assumption does not actually prohibit a database from migrating objects, it just forces us to represent the migration in a specific way: if in the real system object x migrates, then in our model two objects x_1 and x_2 represent x, respectively before and after the migration. The migration itself is represented by the destruction of x_1 and by the creation of x_2.

If two or more objects successively exist at the same address (the first object is deleted, and later the second one is created), we consider them as being the same object. This convention implies that

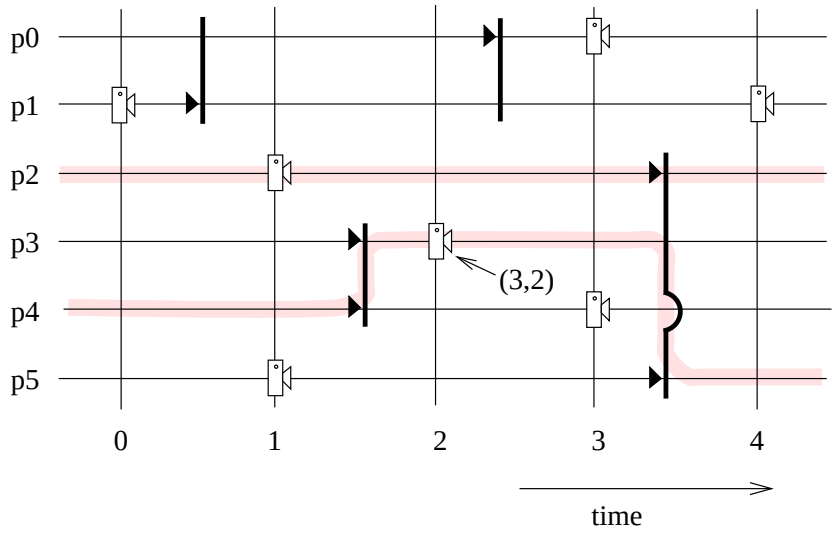

Figure 1: Example execution of a database.

we accept the possibility that an object gets successively created and deleted many times. With this convention, for every pointer value p, the object pointed to by p (noted $*p$) is uniquely defined, even if we do not know at what time p exists. This is essential for keeping our reasonings and proofs simple.

2.3 Graphical notation

Let us explain how we graphically represent executions of database systems. Figure 1 gives an example. Time flows from left to right. Each part is represented by a thin horizontal line. Each transaction is considered as a null-duration event and represented by a thick black vertical line. If a transaction reads a part, the corresponding lines cross; if it also writes the part, an arrow is drawn at the crossing. If a transaction does not access a part, the corresponding lines do not cross; if, however, they must cross for graphical reasons, a little half-circle is drawn at the crossing. For example, the rightmost transaction on the figure reads and writes parts 2 and 5, reads part 3, and does not access any other parts.

When talking about a database execution, we use a special real-valued global clock called the *transaction clock*. This clock takes value 0 at some time before the first transaction, then, in an execution including n transactions, it takes each integer value $t \in [1\,..\,n-1]$ at some time between the t-th and the $t+1$-th transaction. Value n is taken at some time after the n-th transaction. For every t, during the t-th transaction the value of the clock is strictly included between $t-1$ and t.

We use the transaction clock exclusively for the purpose of conveniently describing executions of database systems. We do not assume that transactions have access to this clock, or to any clock at all.

For now, two elements in Fig. 1 are left unexplained: cameras and very thick gray lines; their meaning is explained in Sections 2.6 and 3, respectively.

2.4 Rules about reachability

We use the model of reachability which corresponds with navigational and object-oriented databases. This means that in order to access an object, a transaction must first have access to a pointer to it. Pointers are created, handled and used according to the following rules.

1. There is a fixed set of indestructible objects called *roots*. Pointers to roots are system constants, to which transactions always have access. Transactions always have access to the special pointer value 0, which points nowhere.
2. When a new object is created by a transaction, the transaction gains access to a pointer to it.
3. If a transaction has access to a pointer, it can gain access to the pointed-to object, either for reading only or for reading and writing. Before gaining access to the object, the transaction must lock the part to which the object belongs. Access to an object includes access to the pointer fields in it (read access to fields implies that, recursively, the transaction may gain access to objects pointed to by the fields).
4. When a new object is created, all its pointer fields are set to 0.
5. A pointer field in an object can be written by a transaction which has write access to the object; the value stored is a pointer to which the transaction has access according to rules 1–3.
6. The destruction of an object is considered as a special case of access for writing, *i.e.* can only take place in a transaction which has access to the object and which locks for writing the part to which the object belongs.

This list exhaustively enumerates the operations which can be performed on pointers. For example, we do not allow a transaction to use pointers inherited from another transaction which was executed previously in the same context. We do not allow pointer values created by pointer arithmetic (*e.g.* by adding offsets to pointers as this is frequently done in C programs), pointers to fields inside objects, or pointers stored in the database in places other than pointer fields of objects. These restrictions are necessary for the garbage collection to operate correctly. They are usually enforced in object-oriented DBMS.

Rule 6 expresses the fact that our model permits user code to explicitly delete objects. It opens the way for an anomaly: a pointer may be *dangling*, *i.e.* may point to a location at which there is no object; whenever a reachable objects is deleted, pointers to it become dangling.

An object is said to be *reachable* iff it exists and transactions can access it:

Definition 1 (reachability in executions) *An object is* reachable *in execution E at time t iff it exists at time t and rules 1–6 allow the next transaction which takes place after time t to access x.*

This definition can be equivalently expressed as follows.

Definition 2 (reachability in executions) *The* reachable *objects in execution E at time t form the smallest set such that*

1. *roots are reachable*
2. *and if at time t object x' is reachable and object x exists and x' contains a pointer to x, then x is reachable at time t.*

The equivalence between Definitions 1 and 2 directly results from rules 1–6. Definition 2 can, in turn, be translated into the following non-recursive form.

Definition 3 (reachability in executions) *Object x is reachable in execution E at time t iff there exists an integer $d \geq 0$ and objects $x_0, ..., x_d$ which at time t exist in E and satisfy the following conditions: x_0 is a root; $x_d = x$; and for every integer i satisfying $0 \leq i < d$, object x_i contains a pointer to x_{i+1}.*

The simple proof of equivalence between Definitions 2 and 3 is left out.

An object which exists but is not reachable is called *garbage*. Rules 1–6 imply that once an object is garbage, it is guaranteed to stay garbage forever: a garbage object cannot be destroyed by the system because the hypotheses to apply rule 6 are not satisfied. Garbage objects may (and, to avoid wasting memory, should) be destroyed by the garbage collector.

2.5 Assumptions about garbage collection

The garbage collector is not bound by rules 1–6 above; otherwise, it would be unable discover and destroy unreachable objects.

We assume that every object contains information sufficient to locate pointers in it. This property is required by most, if not all, GCs described so far. It holds in object-oriented database systems (see *e.g.* O_2 [7] and ObjectStore [16]).

We consider *mark-and-sweep* garbage collectors, which divide their work into two clearly distinct phases, respectively called *marking* and *sweeping*. While marking, the GC determines which objects are reachable. The name "marking" is used because some GCs actually mark (by setting a bit inside) the objects as they are found to be reachable.[3] With the possible exception of setting marks in objects, the marking phase involves no modifications in the underlying system; for example, objects are not moved, and garbage objects are not destroyed. Marking is done by a direct application of the recursive definition of reachability: the GC marks as reachable all the roots, then, recursively, all the objects pointed to from within objects previously marked as reachable.

[3] In the context of databases, it is more efficient to keep a separate list of reachable objects rather than to set marks inside objects.

During the sweeping phase, the collector destroys the objects which have been classified as garbage by the marking phase. Reachable objects are left intact.

When studying GC-consistent cuts, we assume that they are used by a mark-and-sweep collector. Marking is performed in the cut. Sweeping is performed in the actual system, concurrently with the mutator's activity; the latter is possible because sweeping, unlike marking, has only minimal synchronization requirements; this question is discussed in detail in Section 4.

We choose to only consider mark-and-sweep GCs, because we believe that they are the best choice for databases. Two other garbage collection methods might be considered: reference counting, and copying collection. With reference counting, each non-root object has an associated counter, which stores the number of pointers currently referencing the object. The object is declared garbage and destroyed when the counter becomes null.

Reference counting is incomplete, *i.e.* does not guarantee the destruction of all garbage objects. For example, if two garbage objects contain pointers to each other, the corresponding counters are non-null, and the objects are not eligible for destruction. This lack of completeness is a fundamental drawback of the method. Another drawback consists in the obligation for the mutator to update a counter in the pointed-to object whenever a pointer is updated.

Because of these drawbacks, the use of reference counting is limited to situations where other methods are especially hard to implement. As far as we know, such situations only arise in distributed systems, in connection with objects susceptible of being referenced remotely. Techniques derived from reference counting are indeed often used in distributed garbage collection; see the survey by Plainfossé and Shapiro [19].

Copying garbage collectors [13, 6, 4], also called *scavengers*, use two memory regions, respectively called *from-space* and *to-space*. The from-space is the memory which is already in use by the system when the GC begins its operation. The to-space is initially empty; pages in this space are allocated progressively, during the operation of the GC.

The reachable objects are copied from the from-space to the to-space. Once the copying is complete, all the pages composing the from-space are deallocated. Garbage objects are not processed in any specific way: their destruction is implicit, it results from the facts that they are not copied to the to-space, and that all the pages in the from-space are deallocated.

In many contexts, scavenging is a valuable garbage collection technique, because by copying reachable objects, the scavenger can diminish fragmentation of empty areas in memory, and improve locality. Moreover, since scavengers only process reachable objects, and avoid the burden of individually deleting garbage objects, they exhibit good performance whenever garbage objects significantly outnumber reachable objects.

We believe, hovever, that scavengers are generally inappropriate for databases. The first reason for this is that most database management systems implement sophisticated policies for placing objects. These policies solve the problem of memory fragmentation, and ensure good locality. In this context, by copying objects a scavenger would not accomplish anything useful, but instead would interfere with these policies. The second reason is that garbage objects in a database are usually much less numerous than reachable objects. We expect that scavenging, which avoids processing garbage objects at the cost of applying a costly operation (namely, a copy) to every reachable object, would perform poorly in these conditions.

2.6 Cuts

When the contents of part i at time t is recorded for the needs of garbage collection, the recording is called a *snapshot* and denoted (i, t). Since we consider transactions as atomic events, we only take into account the possibility of taking snapshots between transactions, *i.e.* at integer times: for (i, t) to be a snapshot, t must be an integer. On figures, snapshots are represented by cameras. For example, Fig. 1 shows several snapshots, among which the snapshot of part 3 at time 2 is pointed by an arrow.

A set of snapshots containing at least one snapshot of each part is called a *cut*.[4] A cut is called *simple* iff it contains one and only one snapshot of each part; otherwise, it is called *multiple*.

We define the *time interval* of a cut to be the interval from the time when the first snapshot in the cut is taken, to the time when the last snapshot is taken, inclusively. If an event happens during the time interval of a cut C, we simply say that it happens *during* C.

In order to mark (*i.e.* to verify the reachability of objects in) a simple cut, we proceed exactly as if the cut was a current state of the system at some time t. In other words, to define reachability in a simple cut, we substitute in Definitions 2 and 3 the words "at time t" with words "in cut C."

When the cut is not guaranteed to be simple, it can contain several copies of a given object, and these copies need not be identical. Moreover, for a given object x, the cut can contain copies of $P(x)$ such that x is present in some of them and absent from others. Therefore, in the general case the definition of reachability in cuts needs to be somewhat more complicated. We use the following definition.

Definition 4 (presence; inconsistent presence; reachability in cuts) *Let C be a cut. An object x is* present *in C iff C contains a snapshot of $P(x)$ where x is present,* i.e. *which contains a copy of x; otherwise, x is* absent *from C. x is* inconsistently present *in C iff C contains both a snapshot of $P(x)$ where x is present and a snapshot of $P(x)$ from which x is absent.*

Objects reachable in C form the smallest set such that

1. *roots are reachable in C*
2. *objects inconsistently present in C are reachable in C.*
3. *if object x is reachable in C and a copy of x present in C contains a pointer to object y and y is present in C, then y is reachable in C.*

Note that in a simple cut, no object can be inconsistently present.

Definition 5 (garbage in cuts) *An object x is* garbage *in cut C iff it is present in C and is not reachable in C.*

[4] This definition of the term *cut* is significantly different from the one used in the context of distributed systems and of their causal cuts. A cut of a distributed system contains, besides the states of the processes, copies of messages which were in transit when the cut was being taken. A cut of a database execution, as defined here, contains only the contents of parts (which correspond with processes in a distributed system), but not the information exchanged between parts during transactions (this would correspond to messages). Moreover, we allow a cut of a database to contain many snapshots of a given part, while a cut of a distributed system may contain only one state per process.

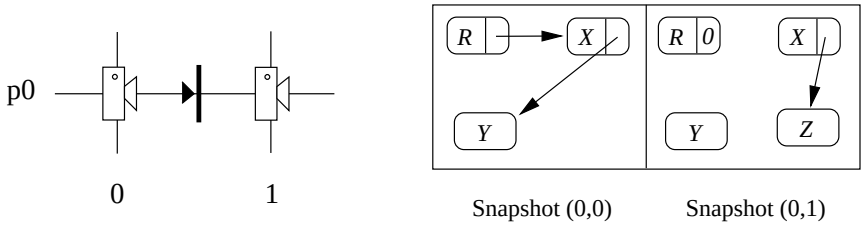

Figure 2: An example explaining rule 3 in Definition 4.

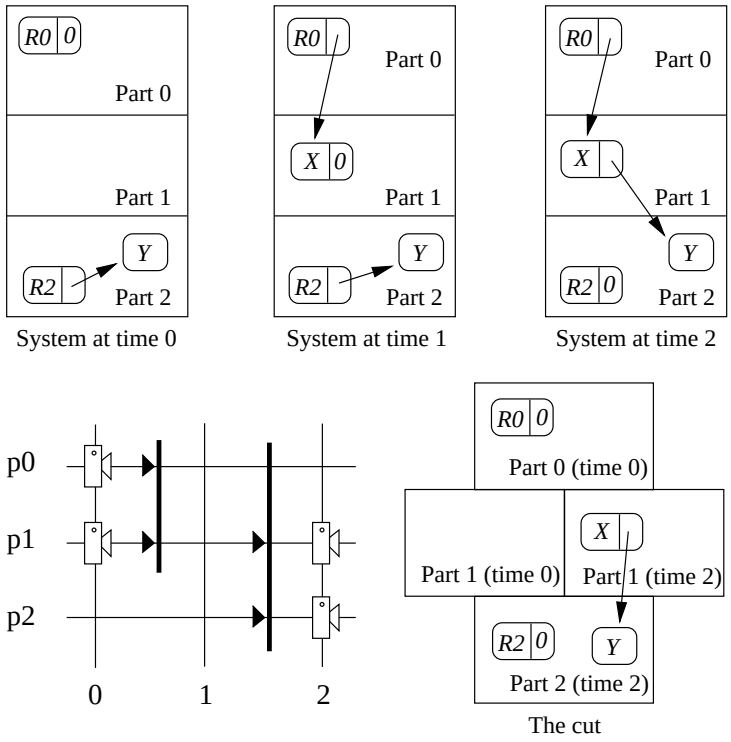

Figure 3: An example explaining rule 2 in Definition 4.

Fig. 2 illustrates rule 3 in Definition 4: objects Y and Z are both reachable in the cut because X is reachable, and copies of X point respectively to Y and to Z (object R is a root).

Rule 2 in Definition 4 says that if an object x is inconsistently present in a cut C, then x is reachable in C. This rule may seem strange. To explain its rationale, let us observe that in order to be inconsistently present in C, x needs to have been created or deleted during C. This, in turn, implies that some transaction had access to x during C. x was therefore reachable at some time during C.

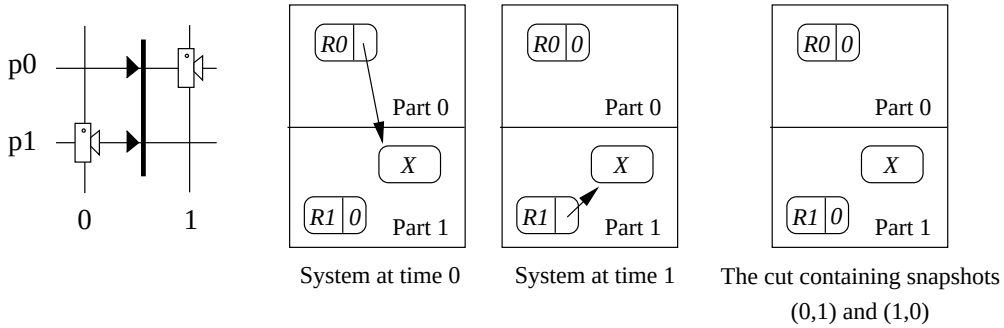

Figure 4: An example inconsistent cut

Therefore, for C to be a faithful representation of what happened in the actual system, x should be reachable in C.

In order to understand the rule better, consider the example cut in Figure 3. Object X and, indirectly, object Y are reachable in the cut because of rule 2 (R is a root object). In the actual system, Y is constantly reachable and X is never garbage (it is initially nonexistent, and later reachable). It is therefore better to declare these two objects reachable in the cut, as we do by applying rule 2, than to declare them garbage, as we would do without the rule.

Like Definition 2, Definition 4 can be rewritten non-recursively.

Definition 6 (reachability in cuts) *Object x is reachable in cut C iff there exists an integer $d \geq 0$ and objects $x_0, ..., x_d$ such that: x_0 is a root or is inconsistently present in C; and $x_d = x$; and for every integer i satisfying $0 \leq i \leq d$, a copy of x_i is present in C and, except for $i = d$, contains a pointer to x_{i+1}.*

Definitions 4 and 6 are equivalent. The simple proof of this fact is left out.

2.7 Consistency criteria for cuts

An object which remains constantly reachable in an execution may appear as garbage in a cut. Fig. 4 shows an example of this anomaly. Let us define a category of cuts in which this anomaly cannot happen.

Definition 7 (cuts containing no false garbage) *A cut C of database execution E* contains no false garbage *iff every object which, in E, is never garbage during C, is not garbage in C.*

Instantaneous cuts, i.e. cuts in which all the snapshots are taken simultaneously, contain no false garbage, because every instantaneous cut is a faithful copy of the state in which the system was when the cut was taken. On the other hand, the example cut in Fig. 4 contains false garbage (namely, object X).

Let us explain why this consistency criterion is important. As we already mentionned in Section 2.4, we assume that cuts are used for garbage collection, in the following way: the collector

marks for destruction the objects which are garbage in a cut C of some system, then deletes the marked objects from the system. If C contains no false garbage, then every object x which is garbage in C is guaranteed to have been garbage in the execution at some time during C. Then, according to remarks we made in Section 2.4, x is still garbage when it gets deleted by the collector. The absence of false garbage in a cut causes therefore the garbage collector to be safe, *i.e.* to delete only garbage.

Similarly, we define a consistency criterion which, if met, ensures that every object which is garbage when the collection starts, will be deleted by the collector. In other words, the criterion ensures the *completeness* of garbage collection.

Definition 8 (cuts exhibiting all garbage) *A cut C of E* exhibits all garbage *iff every object which, in E, remains constantly garbage during C, is garbage in C.*

3 GC-consistent cuts: definition and fundamental properties

In this section we introduce GC-consistent cuts. We state two theorems which imply that such cuts can be used for detecting garbage in a concurrent garbage collector; the proofs of these theorems are deferred to Appendix A.

Definition 9 (path) *Let E be an execution of a database, comprising n transactions; we assume that the database contains m parts. A* path *in E is a function H which goes from the set of integer times of the transaction clock to the set of parts*

$$H \quad : \quad \{0, ..., n\} \rightarrow \{0, ..., m-1\}$$

and which satisfies, for every $t > 0$ belonging to its domain, one of the following conditions:

1. $H(t) = H(t-1)$
2. *or the transaction which takes place between times $t-1$ and t reads part $H(t-1)$ and writes part $H(t)$.*

A path represents the way in which a pointer present at time n in some part i may have been successively copied during E in order to reach this part. According to the definition, $H(t-1)$ and $H(t)$ either are equal (this corresponds with a situation where a pointer value is not copied) or are chosen so that the transaction which happens between times $t-1$ and t has the possibility to copy a pointer from part $H(t-1)$ to part $H(t)$. The latter means that the transaction locks part $H(t-1)$ for reading or writing, and part $H(t)$ for writing.

In Figure 1, two example paths are represented by very thick gray lines (note that many other paths exist in this execution, we just chose to graphically represent two of them). The upper one is straight. This corresponds with a trivial path, which stays in the same part during the whole execution. The lower one shows that a pointer value located in part 5 at time 4 might be there because between times 3 and 4 it was copied there from part 3, after being copied from part 4 to part 3 between times 1 and 2.

Definition 10 (GC-consistent cut) *Let E be an execution of a database. A cut C of E is* GC-consistent *iff it crosses every path,* i.e. *iff for each path H of E there exists some time t satisfying $(H(t), t) \in C$.*

Theorem 1 *Cuts of databases exhibit all garbage.*

This theorem implies that arbitrary cuts can be used for garbage collection, and the resulting garbage collector is complete (although not necessarily safe).

The theorem is remarkable in that it states a consistency property of all cuts, rather than of GC-consistent cuts only or of any other specific class. It implies that a counterexample converse of the one in Fig. 4 does not exist.

Theorem 2 *GC-consistent cuts contain no false garbage.*

Theorems 1 and 2 imply together that a garbage collector which uses GC-consistent cuts can be both safe and complete. The theorems are established in Appendix A.

4 Building and using GC-consistent cuts

The objective of this section is to show that GC-consistent cuts can be built at low cost, and be used as an efficient means of synchronization between the mutator and a concurrent garbage collector. To prove our point, we describe the principles of operation of a garbage collector based on GC-consistent cuts. Our collector builds a GC-consistent cut of the database, and simultaneously checks which objects are garbage in the cut. Then, the objects which are garbage in the cut are deleted from the system. The collector contains three elements: the *cutting agent* which builds the cut, the *marking agent* which assesses the reachability of objects, and the *sweeping agent* which deletes garbage. The cutting agent and the marking agent run concurrently with each other and with the mutator. The sweeping agent begins its operation once the two other agents have finished; it runs concurrently with the mutator.

The agents are described in Sections 4.1–4.3. Section 4.4 discusses selected questions related to performance. Section 4.5 summarizes the results.

4.1 The marking agent

4.1.1 The rules

In order to determine which objects are garbage in the cut, the agent performs the marking phase of the mark-and-sweep algorithm, described in Section 2.5 (page 8). It proceeds according to Definition 4, from which the following rules result.

1. Roots are reachable;
2. objects inconsistently present in the cut are reachable (objects can be inconsistently present only in a multiple cut; if the cut is known to be simple, this rule does not need to be taken into account);

3. if some copy present in the cut of a reachable object x contains a pointer to object y, then y is not garbage, *i.e.* it is either reachable or absent from the cut;

4. the objects which are present in the cut, yet cannot be proven not to be garbage by a recursive application of the three rules above, are garbage.

4.1.2 Three color marking

Rule 3 implies that the agent must visit all the reachable objects in order to discover pointers inside. To determine which objects need to be visited, the agent uses a method based on *three color marking*, a principle introduced by Dijkstra *et al.* [9]. According to Dijkstra's principle, at any time during the marking phase, every object has one and only one of three colors, which have the following semantics.

black The object is known to be reachable and has already been visited.

gray The object is known to be reachable (and therefore needs to be visited), but has not been visited yet.

white We do not know whether the object is reachable.

Classically, the tables of black and gray objects are implemented as hashtables, and there is no explicit representation of the set of white objects: every object which is present in the system and is not gray or black, is white. In our scheme, however, there is an explicit table of white objects. Unlike Dijkstra, we allow objects either to have one color, or to have no color at all, *i.e.* to be absent from all three color-related tables.

The rationale for explicitly distinguishing between white objects, which are garbage, and colorless objects, which are absent from the cut and thus not garbage, is that the sweeping agent is expected to delete only the former. And this agent cannot directly verify which objects are absent from the cut, because when it operates, the cut may no longer exist (let us recall that the sweeping agent operates once the two other agents have finished).

On the other hand, for objects known not to be garbage, we do not need to store information saying whether they are reachable or absent from the cut.

These remarks lead us to depart somewhat from Dijkstra's initial scheme, and to attach the following semantics to colors.

black The object is known not to be garbage and has been visited.

gray The object is known not to be garbage and has not been visited yet.

white The object is present in the cut and is not known to be reachable.

colorless The object does not satisfy the conditions to have any of the three colors, *i.e.* it is not known to be present in the cut or not to be garbage.

Once the marking agent has terminated, all objects which needed to be visited have been, and all objects non-garbage (respectively, present) in the cut have been discovered as such. Therefore, after the termination of the agent no gray objects remain, and the following semantics are attached to the other colors.

black The object is not garbage.

white The object is garbage, and, as such, should be deleted by the sweeping agent.

colorless The object is absent from the cut.

4.1.3 Remarks about the implementation of rules 1–4

Since the cutting agent and the marking agent run concurrently, the marking agent determines which objects are garbage in a cut while this cut is in the process of being built. This leads to two difficulties.

First, the marking agent may need to examine the contents of a part which does not yet have a snapshot. To solve this difficulty, we implement a simple mechanism for communication between the marking agent and the cutting agent. Before visiting a part i, the marking agent sends a request to the cutting agent, asking which snapshots of i have been taken so far. The cutting agent responds with a list of snapshots; if no snapshots of i exist yet, the cutting agent first takes a snapshot, and only then responds. The marking agent is blocked while awaiting the response.

The second difficulty only appears if the cut is multiple (cuts are simple or multiple, depending on the policy used by the cutting agent). In this case, the marking agent has no way to know what snapshots are going to be added to the cut in the future. Thus, when the agent needs to examine the contents of a part, it cannot wait until it gets sure that all snapshots of the part already exist. Instead, it examines the part progressively: at an arbitrarily chosen time it examines all snapshots which exist at that time, then whenever an extra snapshot is taken, it makes a complementary examination of this snapshot.

Keeping in mind these remarks and the semantics which we chose for the colors, let us describe how rules 1–4 are implemented. To implement rule 1, the marking agent colors roots in gray when it starts. Rule 2 causes the agent to enumerate the objects which have to be declared non-garbage because they are inconsistently present in the cut: every object which becomes inconsistently present and is currently white or colorless, *i.e.* not yet declared not to be garbage, must be marked gray. This implies no specific action when the first snapshot of a part is taken, because a part with only one snapshot cannot contain inconsistently present objects. When a subsequent snapshot of a part i is taken, the agent declares non-garbage all the objects in i which become inconsistently present because of this snapshot, *i.e.* which either are present in all the previously-taken snapshots of i and are absent from this snapshot, or on the contrary are absent from all the previous snapshots, and are present in this one. In terms of colors, the agent behaves as follows: whenever a subsequent snapshot of a given part i is taken, the agent colors in gray all the previously white objects in i absent from the new snapshot, and all the previously colorless objects in i present in the snapshot.

To implement rule 3, the agent must visit all reachable objects. In fact, all gray objects are visited (remember that a gray object is not necessarily reachable, it may instead be absent from the cut). There is no harm in visiting the gray objects which are absent form the cut, because visiting such

objects has no effect: the only effect of visiting an object x consists in changing the color of objects pointed from within copies of x; if x is absent from the cut, it has no copies, and visiting x will not result in changing the color of any object.

A gray object can be visited at any time. When visiting object x, the agent processes all the currently existing copies of x, and colors gray the objects which are pointed to from within these copies, and are currently white or colorless. x is then marked black.

It may happen that when an object x is already black, *i.e.* has already been visited, an extra snapshot of $P(x)$, containing an extra copy of x, is taken. In order to correctly implement rule 3, the marking agent must take into account the pointers contained in this copy. For this purpose, the agent acts as follows: whenever a new snapshot is created, for every copy of a black object present in the new snapshot, the copy is visited, and white or colorless objects pointed to from within it are marked gray.

In order to implement rule 4, the agent needs to know which objects are present in the cut. For this purpose, when the first snapshot of a part is taken, the agent colors in white all the objects which are present in this snapshot and which are currently colorless. No action is taken for the objects which are already gray or black, because those objects are not garbage, and in order to apply rule 4 the agent does not need to know whether they are present in the cut.

Rule 4 requires no special action when a subsequent snapshot of a part is taken. To understand this, it suffices to observe that if, based on the first snapshot of $P(x)$, an object x is declared present in the cut, then adding a second or subsequent snapshot of $P(x)$ cannot cause this presence to be lost. If, on the other hand, x is declared absent because it is absent from the first snapshot of $P(x)$, then either x is also absent from all subsequent snapshots of $P(x)$, or not. In both cases, no specific action is needed: either the initial assessment of presence remains true, or the object becomes inconsistently present, and as such is properly treated, *i.e.* marked gray, by the procedure which implements rule 2.

4.1.4 The algorithm

According to remarks made in sections 4.1.1–4.1.3, the marking agent operates as follows.

1. Initially, roots are gray and all other objects are colorless.
2. When the first snapshot of a part is taken, all the objects present in this snapshot and currently colorless are colored in white.

   This operation must be performed for every part in the system. To ensure this, the marking agent may request the cutting agent to make snapshots of parts which currently do not have one; this can be done in any order and at any time, provided that at the end, every part has at least one snapshot.
3. *(this rule only applies if the cut is multiple)* When a subsequent snapshot of a part i is taken, the agent marks gray all currently colorless objects present in the snapshot and all currently white objects which are absent from the snapshot and which, according to their addresses, belong to i.
4. At any time, the agent may *visit* a gray object x, *i.e.*

- Examine all currently existing copies of x and for each pointer p contained in one of these copies, if the pointed-to object $*p$ is white or colorless, mark $*p$ gray.
- Color x black.

All gray objects must be visited; they can be visited at any time and in any order. We expect the marking agent to choose a policy which minimizes swapping. The most obvious optimization consists in always choosing to visit objects which are currently in central memory; objects currently out of memory are visited, and swapping occurs, only when there are no gray objects in central memory.

5. *(this rule only applies if the cut is multiple)* Whenever a subsequent snapshot of a part i is created, the agent enumerates all the black objects in i. The new copies of these objects are visited, and objects pointed from within these copies are marked gray if they are currently white or colorless.

6. The marking agent terminates when nothing is left to be done, *i.e.* when

   - there are no gray objects;
   - and every part has at least one snapshot (this requirement results from rule 2);
   - and the cutting agent declares that the currently existing set of snapshots is a GC-consistent cut (otherwise, the set has to be augmented with more snapshots so as to become a GC-consistent cut).

   When the marking agent terminates, it causes the sweeping agent to begin operation. The table of white objects, *i.e.* of the objects to be deleted, is sent to the sweeping agent. The cut and the two other color-related tables are deleted.

4.2 The sweeping agent

The sweeping agent begins its work immediately after the end of the operation of the marking agent, *i.e.* once the table of white objects is complete. The agent deletes objects directly from the system, not from the cut. Unlike marking, sweeping can be done directly in the system, concurrently with the mutator, because the sweeping agent operates exclusively on unreachable objects, to which the mutator has no access.

In our scheme, sweeping does not involve a complete scan of the system. The sweeping agent only needs to access white objects, which need to be destroyed. Parts which contain no such objects are not accessed. This is important since we believe that in database applications, only a minority of parts contains garbage to be destroyed by any given run of the GC. We believe that this solution is significantly more efficient than the classical marking and sweeping, which requires every page to be examined at sweep time, and thus to be swapped in at least twice (at least once for marking, and once for sweeping).

4.3 The cutting agent

The cutting agent acts as an interface between the system and the marking agent. Its rôle consists in taking snapshots which, together, form a GC-consistent cut of the system, and in allowing the marking agent to use this cut. In this section, we discuss two issues: the mechanisms for taking snapshots, and the choice of the times at which snapshots are taken. The first issue is dealt with in Section 4.3.1. The second issue is split between Section 4.3.2, where we introduce general methods, and Sections 4.3.3 and 4.3.4, where we describe example policies which follow these methods.

4.3.1 Mechanisms for taking snapshots

In this section, we make three assumptions. First, we consider that the database is divided into pages; this assumption is true in most, if not all, database systems. Second, we assume that the GC considers each page as one part (in the general case, a part can be bigger or smaller than a page, as long as no object needs to be split between parts). Third, we assume that the state of the database, as contained in stable storage, takes into account all the changes brought by already committed transactions, and no other changes. In other words, changes brought by a transaction are written to disk when the transaction commits, but no sooner; changes made by a transaction which aborts are never written. This assumption is consistent with the model defined in Section 2.1 and used throughout this report. It is satisfied by many, but not all, database management systems. For example, it is satisfied by system O_2 [7], on the top of which the authors are currently implementing GC-consistent cuts.

Under these assumptions, taking a snapshot of a part amounts to taking a copy of the corresponding page, as stored in stable storage. In practice, the copy is *virtual*: instead of actually copying the page, we just raise a "copy on write" flag on it. An actual copy is later made if the flagged page is about to be modified by a transaction. Usually, the number of pages modified by the mutator during the marking phase is very small compared to the total number of pages. Most virtual copies made by the cutting agent are therefore never transformed into actual copies, and the whole process is efficient.

Some database management systems, *e.g.* EXODUS [12, 11], break our third assumption, and write into stable storage changes made by non-committed transactions. If a page incorporates such changes, it is called *dirty*. Since a snapshot should take into account only the effects of already-committed transactions, taking the snapshot of a dirty page involves two steps: first, we copy the page out of stable storage, then, using the log, we undo in the copy all the changes brought to the page by transactions not committed yet.

4.3.2 Mechanisms for determining when to take snapshots

To determine when to take snapshots, we use the notion of *captured part*.

Definition 11 (captured part) *Let C be a set of snapshots in some execution E. We say that C* captures *part i at time t iff for every path H in E such that $H(t) = i$, for some time $t' \leq t$ we have $(H(t'), t') \in C$.*

This definition means that C captures part i at time t iff C contains a snapshot, taken at time t or before, of every path H such that H goes through part i at time t. Intuitively speaking, C captures

part i at time t iff every pointer which is present in i at time t is recorded in some snapshot in C taken at time t or before (pointers to roots and pointers to newly created objects are excluded from this rule).

For example, on Fig. 1 (section 2.3, page 6), part 3 is trivially captured at time 2 since snapshot $(3, 2)$ belongs to the cut. Part 1 is captured at time 0 for the same reason, but is not captured at time 1 since at times 0 and 1, no snapshots are taken of the path $\{(0, 0), (1, 1), (1, 2), (1, 3), (1, 4)\}$.

The following lemma explains how the notion of captured part can be used to characterize GC-consistent cuts:

Lemma 1 *A set of snapshots C in execution E is a GC-consistent cut of E iff C captures all the parts of E at its end,* i.e. *at the time when the last snapshot in C is taken.*

The lemma directly results from Definitions 9, 10 and 11. □

As we explain below (Section 4.3.3), the cutting agent monitors in real time which parts are currently captured, and which are not. This information is used in order to decide when to take a snapshot of any given page. The following definition can be used by the agent in order to determine which parts are captured.

Definition 12 (captured part) *Let C be a set of snapshots in some execution E. We say that C* captures *part i at time t iff one or more among the following conditions hold*

(i) $(i, t) \in C$;

or (ii) $t > 0$, *and no transaction writes part i between times $t - 1$ and t, and C captures part i at time $t - 1$;*

or (iii) $t > 0$, *and a transaction T writes part i between times $t - 1$ and t, and for every part i' read by T, C captures i' at time $t - 1$.*

Definitions 11 and 12 are equivalent. The equivalence results from the definition of path (Definition 9); the simple proof is left out. □

Definition 12 can be translated into an algorithm which allows the cutting agent to determine at any time t which parts are currently captured by a set of snapshots C. The algorithm maintains a variable called *captured*, which is a table of booleans indexed on parts. All elements in *captured* are initialized to false. For every integer value t of the transaction clock, including 0, the algorithm updates *captured*. After such an update, *captured*$[i]$ expresses the fact that part i is or is not captured at time t. The update at time t is done as follows:

(a) if a transaction T takes place between times $t - 1$ and t, and there exists a part i' read by T such that

$$\textit{captured}[i'] = \text{false}$$

then for every part i written by T, perform

$$\textit{captured}[i] := \text{false}$$

(b) for every part i which has a snapshot taken at time t, perform

$$\mathit{captured}[i] := \text{true}$$

For every t, step *(b)* must be performed after *(a)*, so that if both steps update an entry in *captured*, the update performed by *(b)* persists.

This algorithm is evidently cheap. The proof that it accurately determines which parts are captured, *i.e.* that it faithfully implements Definition 12, is left out because it is purely technical (there are no real difficulties) and we judge it uninteresting.

4.3.3 A policy for building GC-consistent simple cuts

Let us describe a policy which directs the cutting agent to generate a GC-consistent simple cut. The general idea is that snapshots are taken when necessary, but no sooner. This idea leads the agent to take snapshots in two situations: when the snapshot is needed by the marking agent (this is expressed by rule 2 below), and when it must be included in the cut in order for the cut to be GC-consistent (rule 3 below). The policy consists in the following rules.

1. Initially, no snapshots exist.
2. When the marking agent requests access to a part which does not have a snapshot, take a snapshot of this part.
3. If a transaction which writes a captured part is going to commit, then immediately before it commits, take a snapshot of every noncaptured part read by it.
4. When all parts are captured, notify the marking agent that the snapshots existing now form a GC-consistent cut, and halt (the notification is needed by rule 6 in Section 4.1.4).

Let us show that under this policy the cutting agent halts and that it builds a GC-consistent simple cut. First, observe that while the policy is in force, a captured part cannot become noncaptured. This results from rule 3 in the policy and from Definition 12. The definition implies that a part i which is captured at time $t-1$ may be noncaptured at time t only if the transaction which occurs between $t-1$ and t reads a noncaptured part and writes i. The rule precludes this situation.

To show that the cutting agent halts, we use the fact that, according to rule 2 in Section 4.1.4 and to rule 2 above, the marking agent causes all parts to have snapshots. By Definition 12, a part is captured at the time when a snapshot of it is taken. Every page in the database is therefore captured at some time. Since a captured part cannot become noncaptured again, at some point all parts are captured and, by rule 4 above, the cutting agent halts.

The set of snapshots generated by this policy is a cut because every part has a snapshot taken at some time. To show that this cut is GC-consistent, it suffices to observe that when the last snapshot is taken, all parts are captured, and to use Lemma 1. To prove that the cut is simple, it suffices to show that the policy, and specifically rules 2 and 3, do not provide for taking a snapshot of a part which already has a snapshot. For rule 2, this immedately follows from its text. Rule 3 is restricted

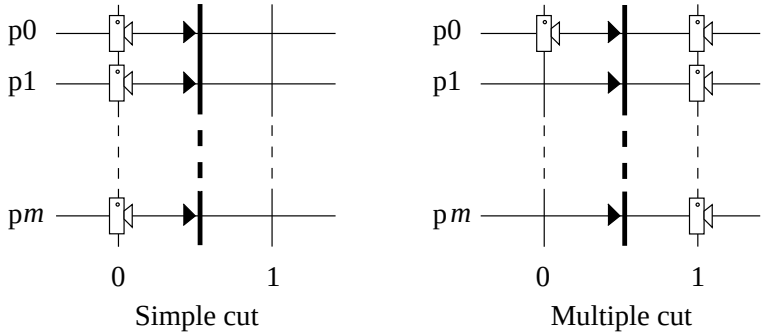

Figure 5: Comparison between a simple and a multiple GC-consistent cut.

to taking snapshots of noncaptured parts. Since, according to Definition 12, a part becomes captured when a snapshot of it is taken, and, as we have already shown, cannot become noncaptured again, rule 3 only provides for taking snapshots of parts which do not yet have a snapshot. □

4.3.4 Building GC-consistent multiple cuts

In this section, we exhibit a simple example showing that GC-consistent multiple cuts are useful, at least in some situations. Then, we quote a set of rules which can be used by the cutting agent in order to build such a cut.

Consider the execution in Figure 5, which comprises one transaction (the execution is shown twice: once in the left part of the figure, and once in the right part). We assume that before the transaction, *i.e.* at time 0, the marking agent wants to examine part 0. The cutting agent is therefore required to take a snapshot of part 0 at time 0. Then, if the cutting agent builds a GC-consistent simple cut, the only cut we can obtain is the one shown in the left part of the figure, where every part has one snapshot, taken at time 0; the straightforward proof of this fact is left out. This cut is expensive to build because the transaction modifies all parts between times 0 and 1, forcing all the snapshots to be implemented as physical, rather than virtual, copies of pages.

If, on the other hand, the cutting agent is allowed to take a GC-consistent multiple cut, it can take the cut shown on the right of the figure. Although this cut looks similar to the previous one, it is much less expensive to build: with one exception, all the snapshots in it are taken after the transaction, and can therefore be implemented by virtual, and not physical, copies.

In order to obtain a GC-consistent cut which is not necessarily simple, we can use the following rules.

1. Initially, no snapshots exist.
2. When the marking agent requests access to a part which does not have a snapshot, take a snapshot of this part.
3. If a transaction writes a captured part, then either

(a) immediately before the transaction commits, take a snapshot of every noncaptured part read by this transaction;

(b) or immediately after the transaction commits, take a snapshot of every part written by the transaction.

4. When all parts are captured, notify the marking agent that the snapshots existing now form a GC-consistent cut, and halt.

Except for rule 3, these rules are identical to the policy described in Section 4.3.3 above. We do not call these rules a policy because they do not say precisely what the cutting agent is supposed to do: rule 3 is nondeterministic, and allows the agent to apply either rule 3a or rule 3b. In order to transform this set of rules into an actual policy, we would need to specify a deterministic criterion according to which the agent has to choose between rules 3a and 3b.

If rule 3a is always chosen, the rules become identical to the policy from Section 4.3.3. For example, the simple cut in the left part of Fig. 5 can be obtained in this way. On the other hand, the multiple cut in the right part of Fig. 5 is obtained by choosing rule 3b.

This set of rules always causes the cutting agent to build a GC-consistent cut, then to halt. In Section 4.3.3 above, we have proven this fact under the policy defined there. Since our set of rules is identical to that policy, except for rule 3, the proof remains valid here, except that we need to prove again the facts which, in the original proof, are established using rule 3. This is the case for only one fact, namely for the fact that once a part is captured, it cannot become noncaptured again. To establish this, we assume on the contrary that part i is captured at time $t-1$ and noncaptured at time t. According to Definition 12, this can only be the case if two conditions are met: *(i)* a transaction T, which occurs between $t-1$ and t, reads a noncaptured part and writes i, and *(ii)* a snapshot of i is not taken at time t. Condition *(i)* cannot be met if rule 3a is applied to T; condition *(ii)* cannot be met if rule 3b is applied instead. Since either rule 3a or rule 3b is applied to T, the two conditions cannot both be met. □

4.4 Remarks about performance

The GC described here is in the process of being implemented as part of O_2, an industrial object-oriented DBMS [7]. We plan to benchmark the GC on realistic datasets, but currently we have no experimental results. We chose not to develop a simplified, fast-to-complete prototype, because we believe that such a prototype would not be capable of running with realistic applications, and would therefore not yield information interesting enough to justify the extra effort.

Since there is no implementation, we do not include here a full discussion of performance. Instead, we only describe three properties which do not depend on the details of the implementation, and which imply that our GC has no serious detrimental effect on the rest of the system. The first property is that a user process cannot be significantly delayed while waiting for a lock held by the GC. In systems where the granularity of locks is the object (*e.g.* O_2, version 5.0 and later), this directly results from the fact that the mutator only locks reachable objects, while the sweeping agent only locks garbage and the other agents in the GC do not lock anything. In systems with page-level locking, like EXODUS [12, 11], ObjectStore [16] or the older versions of O_2 [7], a lock contention

between the sweeping agent and the mutator may occur for a page containing both reachable and garbage objects. Such a conflict cannot, however, significantly delay the mutator because a page is locked by the sweeping agent only for the time necessary to delete garbage in it. This time is short; most notably, while the sweeping agent locks a page, it does not await other locks, and therefore does not risk being blocked.

The two other properties are that the existence of our GC has no effect upon the performance of the system while the GC is not running, and that the GC does not require user code to be compiled in a special way or instrumented. These properties may seem obvious, but in fact they are not satisfied by most concurrent GCs. The reason for this is that most concurrent GCs require user code to be instrumented in order for the write barrier to work: extra code which sends a message to the GC is inserted either before or after every instruction in the mutator that modifies a pointer. For example, with the simplest variant of the write barrier, known as *snapshot at beginning*,[5] while the GC is in operation, the user instruction

```
obj->p = q;
```

may only be executed once the old value of `obj->p` has been sent to the GC. Thus, the instruction is replaced with the following code.

```
if (markingInProgress)
    notifyGC (obj->p);
obj->p = q;
```

where the boolean `markingInProgress` is true while the GC is marking and false otherwise, and the function `notifyGC` sends its argument to the GC.

The need to instrument user code mandates the use of GC-specific compilers, and makes the GC hard to integrate with existing systems. Moreover, the instrumented code is slower than ordinary code, even while the GC is not in operation.

There exist alternative ways to implement write barriers, which do not require user code to be instrumented. For example, the GC can be warned about writes by the virtual memory management hardware [14]. In a database system, the information about pointer modifications is in the log, and the GC can implement a write barrier by reading it from there; this eliminates the need for user code to be instrumented or, more generally, to cooperate with the GC in any specific way. This approach is used by Amsaleg *et al.* [3] and by O'Toole *et al.* [18]. But even with this approach, the existence of the GC has some negative impact on the performance of the system while the GC is not in operation, because the log needs to be examined all the time, not just while the GC is running. Measurements performed in various setups by Amsaleg *et al.* show an overhead of 0.6 % to 5.8 %. O'Toole *et al.* note that the introduction of a garbage collector requires the log to be more complete than it used to be in the previous version of the system (and therefore more costly to maintain); they do not measure the extra cost.

[5] Snapshot at beginning was initially introduced by Abraham and Pattel [1], and is described in detail by Wilson [20].

4.5 Summary

We have described the design of a concurrent garbage collector for databases, based on GC-consistent cuts. The two key elements are the algorithm which concurrently builds GC-consistent cuts, either simple or multiple, and the method for examining a cut which is in the process of being built.

Our GC works in presence of ordinary user code, which does not need to be instrumented. While it is not running, its presence has no effect on the performance of the system. While it runs, it does not lock pages or objects in a way which could block the users for significant amounts of time.

5 A theoretical study of GC-consistent cuts

So far, our approach of GC-consistent cuts was mainly practical: we have explained how and under which assumptions such cuts can be used for garbage collection. In this section, we complete the study by a theoretical analysis of the consistency properties of GC-consistent cuts: we investigate what an observer can learn about an execution by examining a GC-consistent cut of it.

We compare the consistency properties of three classes of cuts: the two classes described so far, namely GC-consistent cuts and GC-consistent simple cuts, and *causal cuts of databases*, introduced below.

To study the consistency of cuts, we use two example questions which can be asked by an observer about a database execution:

— do dangling pointers ever appear during the execution?

— what is the total balance of a group of bank accounts stored in the database?

For each question and for each class of cuts, we verify whether and under which conditions the question can be answered by an observer who looks at a cut of this class instead of looking at the real system.

The section is organized as follows. In Section 5.1, we discuss causal cuts: we recall how causal cuts of distributed systems are defined in previous publications (Section 5.1.1), we exhibit an analogy between distributed systems and databases (Section 5.1.2), and finally we define causal cuts of database executions (Section 5.1.3). In Section 5.2, we state formally the two questions quoted above, and we use them to investigate the consistency of cuts. Section 5.3 summarizes the results.

5.1 Causal cuts

5.1.1 Causal precedence and causal cuts in distributed systems

In this section, we recall the generally accepted definition of causal cuts.[6] Causal cuts are defined in the context of distributed systems. A distributed system is a finite set of processes $p_1, ..., p_n$ which communicate with each other through messages: for every ordered pair of processes (p_i, p_j), a communication channel c_{ij} exists and can transmit messages from p_i to p_j. Channels are reliable,

[6] Footnote 2 on page 4 cites bibliographic references.

i.e. messages are not corrupted, lost or duplicated; we do not care whether channels may or may not deliver messages out of order. The transmission of each message takes a finite, non-null amount of time. We assume that processes do not communicate with each other by means other than messages; they do not communicate at all with the outside world.

The *causal precedence* is a relation defined between the events which happen in processes belonging to a distributed system. Notation

$$e_1 \rightarrow e_2$$

means that e_1 causally precedes e_2. Causal precedence is the smallest partial order such that

1. If e_1 and e_2 are two events occurring in the same process and, according to a physical clock, e_1 occurs before e_2, then $e_1 \rightarrow e_2$ (we assume that two events cannot occur simultaneously in a given process).

2. If e_1 is the transmission of a message and e_2 is the reception of this message, then $e_1 \rightarrow e_2$.

Causal precedence is similar to the ordinary notion of time: using the ordinary time, we say that e_2 happens after e_1 iff e_1 may have influenced e_2 or, to put things differently, iff e_2 takes place in a context in which information generated by e_1 may be present. Similarly, causal precedence is related to the flow of information in a system: $e_1 \rightarrow e_2$ means that there exists a chain of messages through which the process where e_2 occurs has, at the time of e_2, access to information generated by e_1. Conversely, the fact that e_1 and e_2 are incomparable means that each event takes place in a process which currently holds no information coming from the other event.

A *cut* of a distributed system is defined to be a set containing, for every process p_i, one and only one recorded state s_i of this process, called *snapshot*, and for every channel c_{ij}, a *channel state* s_{ij} containing a copy of each message which was sent from p_i before the recording of s_i, and which was not received in p_j before the recording of s_j. The recording of s_i can be considered as an event occurring in p_i (of course, this event does not change the state of p_i). We say that a cut C is *causal* iff the events s_i are incomparable with each other by causal precedence.

Causal cuts are highly consistent because from many points of view, they resemble instantaneous cuts. Intuitively, this results from the resemblance which exists between time and causal precedence: having the events s_i incomparable by causal precedence is like having them simultaneous. It is beyond the scope of this report to describe the exact extent to which causal cuts are consistent, or to explain in a better-than-intuitive way why they are consistent; see [5].

5.1.2 Causal precedence in databases

We consider parts in a database as analogous to processes in a distributed system. The contents of a part corresponds with the state of a process. We define causal precedence in a database execution to be a relation between snapshots, *i.e.* between observable states of parts.

A transaction modifies the contents of the parts to which it has write access. The modifications depend on the previous contents of all the parts to which the transaction has read access. This means that if transaction T writes part i, the snapshot of part i taken immediately after T causally depends on the snapshots of all parts read by T, taken just before the beginning of T.

These remarks lead us to define causal precedence in databases as follows.

Definition 13 (causal precedence in a database execution) *Let E be a database execution. The causal precedence in E is the smallest partial order between snapshots in E such that*

1. *for every part i and for every two integer times t and t', relation $t \leq t'$ implies $(i,t) \rightarrow (i,t')$*
2. *if a transaction taking place between integer times t and $t+1$ reads part i and writes part i', then $(i,t) \rightarrow (i',t+1)$*

This definition poses a problem, due to an important difference which exists between databases, as defined in this report, and distributed systems. In a distributed system, processes are allowed to exchange information only through messages. An analysis of the messages is therefore sufficient for determining if a given event causally precedes another one. In a database execution, information is exchanged between parts during transactions, but this is not all: except for pointers, which must follow the rules listed in Section 2.4, transactions are free to exchange information between each other and with the outside world. For example, transaction T may represent a bank employee checking how much stock you own, and T' may represent the same employee granting you a credit. In this case, the amount of credit granted (represented by data in some part i') may depend on the amount of stock you own (represented by data in part i), even though no transaction accesses both i and i', and, according to Definition 13, snapshots of i are not causally related to snapshots of i'.

The problem described above implies that Definition 13 makes much less sense than the definition of causal precedence in distributed systems. Note, however, that the problem disappears if we assume that transactions do not exchange information between each other or with the outside world. In this case, every execution of a database can be implemented by an execution of a distributed system, and the causal precedence in the database execution will match the causal precedence in the implementing distributed system. We do not quote a complete proof of this, but we affirm that such a proof exists, and is constructive.

5.1.3 Causal cuts in databases

Our definition of causal cuts in databases is based on causal precedence as introduced by Definition 13.

Definition 14 (causal cuts of in databases) *A cut C of a database execution E is causal iff*

1. *any two different snapshots (i,t) and (i',t') belonging to C are causally incomparable*
2. *if a transaction which takes place between times t_0 and t_0+1 reads part i and writes part i', then at least one of the following holds*
   (a) *the snapshot of i in C is taken at or before time t_0*
   (b) *or the snapshot of i' in C is taken at or after time t_0+1.*

Condition 1 in the definition implies that every causal cut is simple. This, in turn, implies that expression "the snapshot of i in C," used in condition 2, determines a unique snapshot.

This definition is to a large extent analogous to the definition of causal cuts in distributed systems. At first sight, there are two problems with the analogy. First, a causal cut of a distributed system includes *channel states*, which seem to have no counterpart in the causal cuts defined here. Second, condition 2 in our definition appears to be a restriction bearing no analogy with causal cuts in distributed systems. In reality, condition 2 and the absence of channel states are related: the former compensates for the later.

To explain this, let us recall that channel states contain all the messages sent from a process before the state of this process is recorded, and received in another process after the state of that process is recorded. In the context of databases, there are no messages. Instead, information is exchanged between parts by transactions. A transaction which reads part i and writes part i' is assumed to transmit information from i to i'. To have a counterpart to channel states, for every causal cut C we would therefore need to implement the following rule:

> If a transaction which takes place between times t_0 and $t_0 + 1$ reads part i and writes part i', and the snapshot of part i in C is taken at time $t_0 + 1$ or later, and the snapshot of part i' in C is taken at time t_0 or before, then information transmitted by the transaction from i to i' must be recorded as part of C.

This rule requires us to examine information transmitted from one part to another during a transaction. But our approach precludes this: we consider transactions as atomic events, and refuse to examine what happens inside. Fortunately, there is a special way of implementing the rule without looking inside transactions. Instead of recording information transmitted, we simply outlaw the situations in which the recording needs to be made. This amounts to imposing an extra requirement on the times at which snapshots in C are taken. This is what does condition 2 in Definition 14.

Causal cuts in databases can be characterized as follows.

Theorem 3 *A cut C of a database execution E is causal iff it crosses every path H of E in exactly one point,* i.e. *iff for every path H, there is one and only one time t such that $(H(t), t) \in C$.*

The proof of this theorem is deferred to Section A.6.

5.2 The consistency of the three classes of cuts

In this section, we discuss the classes of cuts of databases introduced above: causal cuts, GC-consistent simple cuts, and GC-consistent cuts. We investigate their consistency properties with respect to the two questions introduced in the beginning of Section 5.

Every causal cut is also a GC-consistent simple cut (this results from Definition 10 and Theorem 3). Therefore, causal cuts satisfy at least all the consistency properties satisfied by GC-consistent simple cuts. An important question is whether causal cuts are "strictly more consistent" than GC-consistent simple cuts in any meaningful way. This question is addressed in Section 5.2.1. Similarly, every GC-consistent simple cut is also a GC-consistent cut, and therefore GC-consistent simple cuts are at least as consistent as GC-consistent cuts. We want to know whether GC-consistent simple cuts are strictly more consistent than GC-consistent cuts; Section 5.2.2 deals with this question.

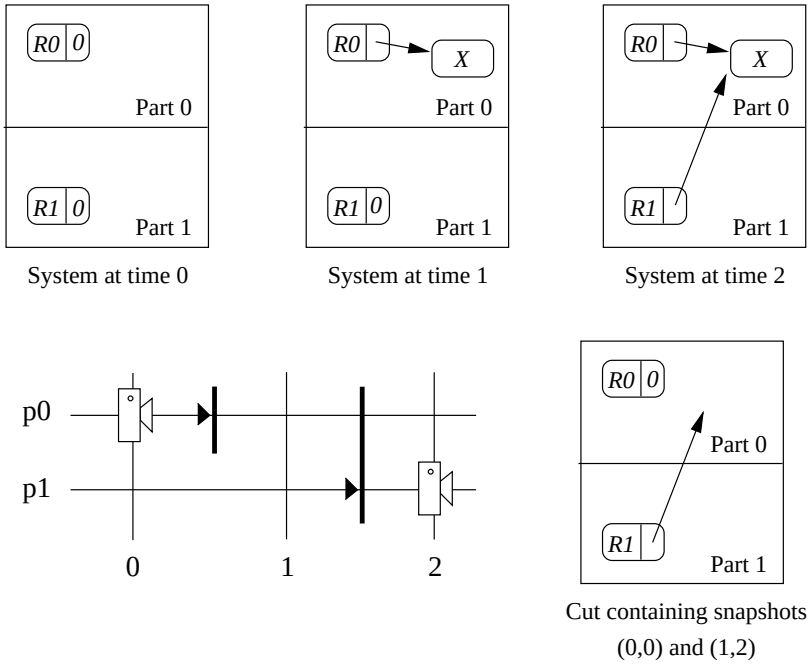

Figure 6: A pointer is dangling in a GC-consistent simple cut although it is not dangling at any time in the underlying execution.

5.2.1 The detection of dangling pointers

Let us recall that in a database execution, a pointer is *dangling* when it points to an object which currently does not exist. The existence of dangling pointers is usually considered to be an inconsistency in the system. An observer may want to know whether dangling pointers appear during an execution. In a cut C, a pointer is considered as dangling iff it is present in C, but the pointed-to object is not present in C.

A GC-consistent simple cut may contain a dangling pointer even if such pointers never appear in the underlying execution, and even if no objects are destroyed during the execution (to undestand why we quote the latter condition, compare this sentence with Theorem 4 below). An example of this possibility is shown on Fig. 6. In this figure, object X is created in part 0 between times 0 and 1, and a pointer to X is copied to part 1 between times 1 and 2. The cut contains a dangling pointer to X because it includes a snapshot of part 0 taken when X does not exist yet and a snapshot of part 1 taken when this part already contains a pointer to X. Since the cut in Fig. 6 is GC-consistent and simple, and no objects are destroyed in the corresponding execution, this example proves our point.

The following theorem implies that an inconsistency similar to the one from Fig. 6 cannot happen with a causal cut.

Theorem 4 *Let E be a database execution in which no pointer is ever dangling and during which no objects are destroyed. Then, no pointer is dangling in a causal cut of E.*

The proof is deferred to Section A.7.

This theorem implies that in a system which does not destroy objects, we can use causal cuts to try to detect dangling pointers in an execution: the detection may be incomplete (*i.e.* dangling pointers may remain undetected), but it does not issue false alarms.

It results from the example and from the theorem that GC-consistent simple cuts are strictly less consistent than causal cuts.

5.2.2 Total balance of several bank accounts

Let v be a variable in the system. We assume that v constantly exists during a given execution E. The value of v at time t is noted v^t. The value of v in the simple cut C is noted v^C. To define v^C, we observe that C contains one and only one snapshot of $P(v)$ (the part containing v), and we say that v^C is equal to the value of v as seen in this snapshot. We do not define v^C in the case where C is a multiple cut.

Consider an execution E of a database containing integer variables $v_0, ..., v_l$. We consider that each variable represents a bank account. Money can be transferred between the accounts, but the total balance is assumed to remain constant: the value

$$S = \sum_{j=0}^{l} v_j^t$$

does not depend on t. The following theorem implies that S can be computed in a straightforward manner using a GC-consistent simple cut of E.

Theorem 5 (sum of bank accounts) *Let E be an execution of a database which contains several integer variables $v_0, ..., v_l$. We assume that the sum of the variables remains constant,* i.e. *that there exists an integer S such that the statement*

$$S = \sum_{j=0}^{l} v_j^t$$

holds for every integer time t. Let C be a GC-consistent simple cut of E. Then,

$$S = \sum_{j=0}^{l} v_j^C \tag{1}$$

The proof of this theorem is quoted in Section A.5.

There is no equivalent of this theorem for GC-consistent cuts, *i.e.* there is no general method to compute the value S using a GC-consistent cut of E. To prove this, we use the following counterexample. We define E and F to be two executions, which are identical except for the values

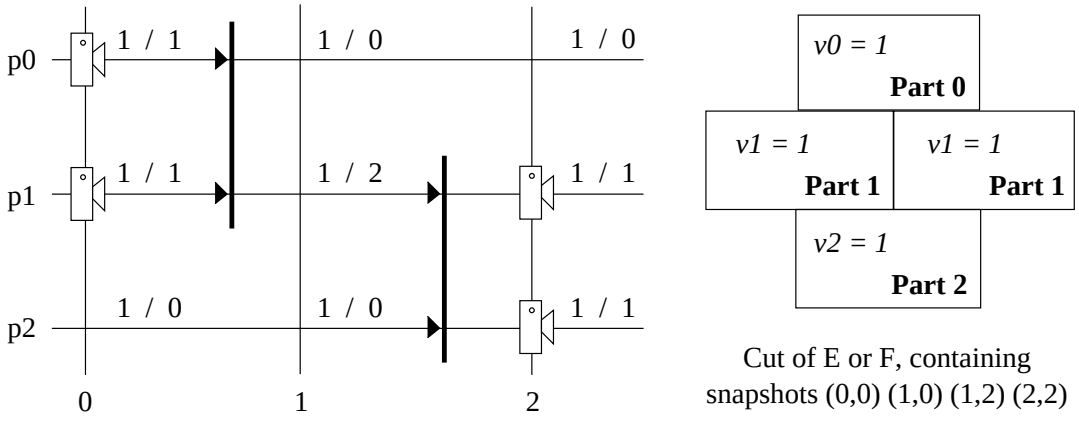

Figure 7: Two executions have different sums of variables, but their GC-consistent cuts are identical.

taken by their respective variables. Each execution operates on a system comprising three parts and containing one variable per part (for $i = 0, 1, 2$, part i contains variable v_i). In each execution, the sum of the three variables remains constant; it is equal to 3 in E and to 2 in F.

Fig. 7 represents the executions. It contains slash-separated pairs of numbers. Each pair corresponds with an integer time and with a part. The two numbers represent the value taken at the corresponding time by the variable contained in the corresponding part, in E and in F respectively (in this order). For example, at the crossing between the line representing part 2 and the one representing time 0, we have the pair "1 / 0," which means that v_2 takes, at time 0, value 1 in E and value 0 in F.

A GC-consistent cut of E or F is represented on the figure (it makes sense to talk about a cut of "E or F" because E and F are almost identical). The snapshots in the cut (especially, the values of the variables, represented in the right part of the figure) are the same regardless of whether the cut is taken for E or for F. Thus, by examining the cut we cannot decide whether it is a cut of E or a cut of F. Since the sums of variables in E and in F are different, this proves our point. □

These results show that GC-consistent simple cuts can be used to compute the total balance of a sum of bank accounts, assuming that the sum does not change while the cut is being taken. GC-consistent cuts cannot generally be used for this purpose. These facts imply that GC-consistent cuts are less consistent that GC-consistent simple cuts.

5.3 Summary of results about consistency

We have introduced a new class of cuts, the causal cuts of databases. These cuts are analogous to causal cuts of distributed systems. We have compared the consistency properties of causal cuts (defined in the context of databases), of GC-consistent simple cuts, and of GC-consistent cuts. For this purpose, we have used two consistency criteria. The first criterion requires that for every execution E in which no dangling pointers appear and no objects are destroyed, a cut of E contains no dangling pointers (Section 5.2.1). The second criterion requires that if bank accounts are stored in a database

Defined in	Preservation of garbage *Sec. 2.7, Def. 8*	Absence of false garbage *Sec. 2.7, Def. 7*	Sum of bank accounts *Sec. 5.2.2*	Dangling pointers *Sec. 5.2.1*
Causal cuts	×	×	×	×
GC-consistent simple cuts	×	×	×	
GC-consistent cuts	×	×		
Arbitrary cuts	×			

Table 1: Consistency criteria and classes of cuts.

and the total balance of these accounts remains constant during an execution, then this balance can be deduced from a cut of the execution (Section 5.2.2).

Table 1 summarizes the results about consistency of cuts defined in this report, and quoted in Sections 2.7, 3 and 5.2. The table compares the three classes of cuts discussed in this section, and arbitrary cuts. It says which criteria are met by which classes of cuts. A cross in the table means that we have established that all the cuts in a class meet a criterion. Conversely, for every cross missing, we have exhibited a counterexample which shows that not all cuts in the class meet the criterion.

Every causal cut is also a GC-consistent simple cut; this results from Theorem 3 in Section 5.1.3. Similarly, every GC-consistent simple cut is also a GC-consistent cut. These facts and the results summarized in the table allow us to state that GC-consistent cuts exhibit strictly weaker consistency properties than GC-consistent simple cuts, which in turn are strictly less consistent than causal cuts of databases.

6 General summary

We have defined GC-consistent cuts of databases. The essential fact about GC-consistent cuts is that they can be used to *synchronize* the mutator and a mark-and-sweep garbage collector, *i.e.* to allow the mutator and the collector to run concurrently. We have discussed and established the properties of GC-consistent cuts which imply this fact. First, we have shown that a garbage collector can correctly determine which objects should be deleted from the database by examining a GC-consistent cut of the database, and without examining the database itself; this property is stated and discussed in Section 3, and established in Appendix A. Second, in Section 4 we have explained how a GC-consistent cut of a database can be built concurrently (*i.e.* while the database is in operation, without stopping it) and cheaply, and can be used by a GC while it is being built.

We have analyzed the consistency properties of GC-consistent cuts from a theoretical point of view (Section 5). This analysis is based on a comparison between the properties of GC-consistent cuts and those of other kinds of cuts: causal cuts of distributed systems, and two kinds of cuts defined in this article.

A Proofs

This section contains proofs of Theorems 1, 2, 5, 3 and 4, in this order. In Section A.1, we introduce the notation used in all our proofs. Section A.2 contains the relatively simple proof of Theorem 1.

The proof of Theorem 2 is done in Sections A.3 and A.4. Although exclusively based on elementary concepts, the proof is complicated. Let us describe its structure. First, we introduce several concepts and facts necessary for understanding the behavior of GC-consistent cuts. We then proceed by a double induction. The first induction is done on the *duration* of a cut, *i.e.* on the length of its time interval. In order to prove that a given cut C satisfies the theorem, we first observe that C trivially satisfies the theorem if it is an instantaneous cut; then, assuming that C is not instantaneous, we build a cut C' which is similar to C, but has a shorter time interval. We prove that if C' satisfies the theorem, then so does C. For this purpose, we take an arbitrary object x which remains constantly reachable in E during C, and we prove that x is reachable in C. This proof is done by induction on the *reachability index of x in C', i.e.* on the number of pointers which an observer needs to successively dereference in C' in order to reach x, starting from a root or from an inconsistently present object; this distance is finite since the induction hypothesis implies that x is reachable in C'.

Section A.5 contains a proof of Theorem 5. This proof is relatively short. It borrows heavily from the proof of Theorem 2: it is based on definitions and facts introduced in Section A.3, and uses an induction on the duration of GC-consistent cuts, similar to one of the inductions in Section A.4.

Theorems 3 and 4 are established in Sections A.6 and A.7, respectively. These proofs are relatively simple, and are not related to Sections A.2–A.5.

A.1 Notation

We assume that E is an execution which contains n transactions and takes place in a system comprising m parts numbered $0, ..., m-1$. All the cuts mentioned in the appendix are cuts of E. Similarly, all sets of snapshots mentionned are sets of snapshots taken in E. All the paths[7] mentioned are paths in E. The term "clock" is used for the transaction clock of E.[8]

Symbols t, t' and t_a (for any index a) represent integer values of the clock. This implies that, whenever one of these is used, we silently assume that it represents an integer value in the interval $[0\,..\,n]$. For example, the expression "for every t" should be understood as meaning "for every t representing a legal integer value of the transaction clock of E" or, which is equivalent, "for every integer t satisfying $0 \leq t \leq n$."

Symbol i, as well as derived symbols like i' and i_a (for any index a), represents the index of some part and is assumed to always be an integer satisfying $0 \leq i < m$.

Let us define the *reachability index* of an object in a cut. The proof of Theorem 1 (Section A.2) and the second induction in the proof of Theorem 2 (Section A.4) are based on this concept.

Definition 15 (reachability index) *Let C be a cut and let x be an object reachable in C. We define the reachability index of x in C to be the smallest positive integer d for which there exist objects*

[7] Paths are introduced by Definition 9 in Section 3.
[8] The transaction clock is introduced in Section 2.3.

$x_0, ..., x_d$ such that x_0 is a root or is inconsistently present in C; and $x_d = x$; and for every integer $j \in [0 .. d-1]$, a copy of x_j present in C contains a pointer to x_{j+1}.

Definition 6 implies that this definition is sound, *i.e.* that the reachability index in a cut C is defined for every object reachable in C.

A.2 Proof of Theorem 1

Theorem 1 says that cuts of databases exhibit all garbage. To establish the theorem, it suffices to prove that every object which remains constantly garbage in E during C is garbage in C, *i.e. (i)* is present in C and *(ii)* is not reachable in C. To prove *(i)*, it suffices to observe that if an object x is constantly garbage during C, then it also remains constantly existent, and is present in every snapshot of $P(x)$ containted in C; since C contains a snapshot of $P(x)$, object x is present in C and *(i)* holds.

We establish *(ii)* by contradiction: we assume that the set S of objects which are constantly garbage in E during C, yet reachable in C, is nonempty. Let then d be the smallest reachability index of an object in S. Let x be an element of S with reachability index equal to d. We have $d = 0$ iff x is a root or is inconsistently present in C. These two conditions are not met because x is constantly garbage. For the first condition, the reasoning is trivial; for the second one, it suffices to observe that being constantly garbage in E during C, x is constantly existent in E during C, and therefore is present in all snapshots of $P(x)$ belonging to C.

We thus have $d \geq 1$. There exists then an object x' such that some copy of x' present in C contains a pointer to x, and x' is reachable in C with reachability index $d-1$; the proof of existence of x' is easy, it is based on Definition 15; we omit it. Then, by definition of d, we have $x' \notin S$. The definition of S and the fact that x' is reachable in C imply that x' is not constantly garbage in E during C. Since being garbage is a stable property, this in turn implies that x' is not garbage at the beginning of the time interval of C.

Since a copy of x' present in C contains a pointer to x, we know that at some time during C, x' exists and contains a pointer to x. Let t_0 be the earliest time during C when this is the case. Consider two cases, depending on whether t_0 is the beginning of the time interval of C or not. In the first case (*i.e.* if t_0 is the beginning), at time t_0 object x' exists and is not garbage (and, thus, is reachable) in E, and contains a pointer to x. By Definition 4, and because x exists at time t_0, x is therefore reachable in E at time t_0. This is in contradiction with the fact that $x \in S$.

Consider the remaining case. Then, $t_0 - 1$ belongs to the time interval of C. By definition of t_0, object x' exists and contains a pointer to x at time t_0, and does not contain such a pointer at time $t_0 - 1$. This means that the transaction which takes place between times $t_0 - 1$ and t_0 has access to a pointer to x. This, and the fact that x exists at time $t_0 - 1$ allow us to use Definition 1 and state that x is reachable at time $t_0 - 1$. This is in contradiction with the fact that $x \in S$. □

A.3 Concepts and facts about GC-consistent cuts

First, let us state a simple property of paths.

Lemma 2 *Let H' and H'' be two paths satisfying $H'(t_0) = H''(t_0)$. Then, the function*

$$H \quad : \quad \{0, ..., n\} \rightarrow \{0, ..., m-1\}$$

defined as follows

1. *for $t \leq t_0$, $H(t) = H'(t)$*
2. *for $t \geq t_0$, $H(t) = H''(t)$*

is a path.

The lemma results directly from Definition 9. □

Now, let us quote two definitions and one lemma which will be used later in order to transform a GC-consistent cut C into a cut C' similar to C, but having a shorter duration. This transformation is an essential element of the inductive proofs in Sections A.4 and A.5.

Definition 16 (temporal restriction of a set of snapshots) *Let C be a set of snapshots. We call* temporal restriction *after t_0 of C the set C' of snapshots built according to the following rules.*

1. *for every i and every $t < t_0$, we have $(i,t) \notin C'$*
2. *for every i and every $t > t_0$, we have $(i,t) \in C'$ iff $(i,t) \in C$*
3. *for every i, we have $(i,t_0) \in C'$ iff C captures[9] part i at time t_0.*

Lemma 3 *Let C be a GC-consistent cut. The temporal restriction after t_0 of C, called C', is a GC-consistent cut.*

Proof of lemma 3 Under the lemma's hypotheses, C' is a set of snapshots. We need to prove that C' is a GC-consistent cut. According to Definition 10, this reduces to proving two facts: *(i)* C' contains at least one snapshot of every part and *(ii)* C' crosses every path. We first show that *(ii)* implies *(i)*, then we establish *(ii)*. To prove the implication, we define for every i the constant function H_i such that for every t, $H_i(t) = i$. It results from Definition 9 that H_i is a path. This implies that if *(ii)* holds, then C' crosses each H_i. This, in turn, implies that for each i, there exists a t such that $(i,t) \in C'$, and that *(i)* holds. Thus, *(ii)* implies *(i)*.

Now, let us prove by contradiction that *(ii)* holds. We assume on the contrary that C' does not cross some path called H', and under this assumption we will show the existence of a path H which does not cross C. The existence of such a path is in contradiction with the fact that C is a GC-consistent cut.

Since H' does not cross C', snapshot $(H'(t_0), t_0)$ does not belong to C'. From this, from rule 3 in Definition 16 and from Definition 11, we deduce that there exists a path H'' such that

$$\begin{array}{rrcl} & H''(t_0) & = & H'(t_0) \\ \text{for every } t \leq t_0, & (H''(t), t) & \notin & C \end{array}$$

[9] This word is introduced in Definition 11, Section 4.3.2.

We define path H as follows:

$$\text{for } t \leq t_0, \qquad H(t) = H''(t)$$
$$\text{for } t \geq t_0, \qquad H(t) = H'(t)$$

Lemma 2 implies that H is indeed a well-formed path. Let us prove that H does not cross C, *i.e.* that for every t, we have $(H(t), t) \notin C$. This relation holds for $t \leq t_0$ since for such values of t we have $(H''(t), t) \notin C$ and $H(t) = H''(t)$. For $t > t_0$, it holds since we have $(H'(t), t) \notin C'$ and $H(t) = H'(t)$ and, by Definition 16, C and C' coincide for times $t > t_0$. □

Finally, let us introduce two lemmas which enumerate the conditions under which a transaction can gain access to objects and pointers.

Lemma 4 *Let T be a transaction which occurs in E between times t and $t+1$, and let p be a pointer value. We assume that p points to a non-root object $*p$ which either exists at time t or is not created by T.*[10] *T can gain access to p only if there exist an integer $d_p \geq 0$ and objects $x_0^p, ..., x_{d_p}^p$ such that*

1. *x_0^p is root;*
2. *for every integer $j \in [0\,..\,d_p - 1]$, at time t object x_j^p exists and contains a pointer to x_{j+1}^p;*
3. *at time t, object $x_{d_p}^p$ exists and contains a copy of p;*
4. *for every integer $j \in [0\,..\,d_p]$, T locks $P(x_j^p)$.*

Proof The proof is done by contradiction: we assume that transaction T takes place between times t and $t+1$, and has access to one or more pointers which violate the lemma. In this proof, instead of viewing T as an atomic event, we consider its execution as a finite-time suite of events, which can be observed with the help of a real-time clock. Rules in Section 2.4 enumerate the situations in which T can have access to a pointer: T has access to pointers to roots (rule 1) and can progressively gain access to some other pointers according to rules 2 and 3.

Let p represent, among the pointers which violate the lemma, one to which T gains access first. Note that p may not be uniquely defined as T can simultaneously get access to several pointers, making several pointers "first *ex aequo*." In order to violate the lemma, p must satisfy its hypotheses: it must point to a non-root object which either exists before the beginning of T or is not created by T. Let us show that T initially gets access to p through rule 3 in Section 2.4 (*a priori*, rules 1, 2 or 3 could be used). T cannot get access to p according to rule 1 because $*p$ is not root. Access through rule 2 is impossible if $*p$ is not created by T. Otherwise (*i.e.* if $*p$ is created by T), rule 2 can only be applied when T creates $*p$. And, by hypothesis, in this case $*p$ exists at time t. Therefore, before $*p$ can be created by T, it must first be destroyed by T. By rule 6, this implies that T must have access to p before the creation of $*p$, *i.e.* before the time when rule 2 applies. The rule cannot, therefore, give T its initial access to p. This initial access is therefore acquired through rule 3, *i.e.* as part of gaining

[10] As explained in Section 2.2, we consider that a given object can be succcessively created and deleted many times. For example, x may exist at time t, and between times t and $t+1$, transaction T may delete then create x.

initial access to some object x which contains a copy of p. We consider three cases: x is either *(i)* a root or *(ii)* a non-root object created by T or *(iii)* a non-root object not created by T.

In case *(i)*, we set $d_p = 0$ and $x_0^p = x$. This causes conditions 1–4 in the lemma to be satisfied (the trivial verification of this is left out), which implies that p does not violate the lemma. Thus contradiction.

Case *(ii)* is impossible: when T initially gains access to an object created by T itself, p cannot be present in the object as all the pointer fields in it are equal to 0; this results from rule 4.

In case *(iii)*, we observe that before accessing x, T must have had access to a pointer q to x. Since T had access to q before gaining access to p, the definition of p entitles us to assume that q does not violate the lemma. Since q satisfies the lemma's hypotheses (this directly results from the hypotheses for case *(iii)*), we can use the lemma and say that there exist an integer $d_q \geq 0$ and objects $x_0^q, ..., x_{d_q}^q$ satisfying conditions 1–4 in the lemma.

We are now ready to obtain a contradiction by proving that p satisfies the lemma. For this purpose, we define the integer d_p and objects $x_0^p, ..., x_{d_p}^p$ as follows: $d_p = d_q + 1$, $x_j^p = x_j^q$ for $j \in [0\,..\,d_p - 1]$ and $x_{d_p} = x$. This definition causes conditions 1–4 to be met; the verification of this fact is left out as it is somewhat long, yet easy and boring. Therefore, p does not violate the lemma. Thus contradiction. □

A.4 Proof of Theorem 2

Theorem 2 says that GC-consistent cuts contain no false garbage. First, observe that if a cut C contains a snapshot $(P(x), t)$ from which x is absent, then x is either absent from C or, by line 2 in Definition 4, is reachable in C; in either case, x is not garbage in C.

Let C be an arbitrary GC-consistent cut. To establish the theorem, it is sufficient to prove that every object which is never garbage in E during C is not garbage in C. We call t_1 and t_2 the times when, respectively, the first and the last snapshot in C is taken. The proof is done by induction on the duration of C, *i.e.* on the value $t_2 - t_1$. The case $t_2 - t_1 = 0$ is easy: if $t_1 = t_2$, then C only contains snapshots taken at time $t_1 = t_2$, and is therefore an instantaneous cut, which faithfully represents the state of E at time $t_1 = t_2$. This implies that the lemma holds for C in this case.

The rest of the proof deals with the case $t_1 < t_2$. We define C' to be the temporal restriction of C after $t_1 + 1$. By Lemma 3, C' is a GC-consistent cut. C' contains no snapshots taken before time $t_1 + 1$ or after time t_2 (the easy proof of this sentence is omitted). This fact implies that the duration of C' is strictly less than $t_2 - t_1$ and, by the induction hypothesis, C' can be assumed to satisfy Theorem 2. The fact also implies that the time interval of C' is included in the time interval of C. Therefore, every object x which is never garbage in E during C is not garbage in C'. To complete the proof of the theorem, it is therefore sufficient to establish the following proposition.

Proposition 1 *Let x be an object which is not garbage in C'. Then, x is not garbage in C.*

In order to establish proposition 1, we need to state and prove the following proposition.

Proposition 2 *Let x be an object and let t be a time such that $(P(x), t) \in C'$ and x does not exist at time t. Then, x is not garbage in C.*

Proof of proposition 2 Consider an object x and a time t which satisfy the hypotheses of the proposition. To simplify notation, we define

$$i = P(x)$$

Let us show that x is not garbage in C. We are in one of the following cases.

Case 1: $t > t_1 + 1$

Case 2: $t = t_1 + 1$ and $(i, t) \in C$

Case 3: $t = t_1 + 1$ and $(i, t) \notin C$ and at time t_1, object x does not exist in E

Case 4: $t = t_1 + 1$ and $(i, t) \notin C$ and at time t_1, object x exists in E

Let us successively deal with cases 1–4. In case 1, we have $(i, t) \in C$ because $t \geq t_1 + 1$, and therefore C and C' coincide at time t. In case 2, we have $(i, t) \in C$ by hypothesis. Since x does not exist at time t, in these two cases C contains a snapshot of i from which x is absent. According to the remark made at the beginning of this subsection, x is then not garbage in C.

In case 3, we use the fact that, since $(i, t_1 + 1) \in C'$, cut C crosses at time $t_1 + 1$ or before every path H such that $H(t_1 + 1) = i$. More precisely, since C contains no snapshots taken before time t_1 and because of the assumption that $(i, t_1 + 1) \notin C$, cut C crosses every such path at time t_1. In particular, C crosses at time t_1 the constant path H_i such that for every t, $H_i(t) = i$. This means that $(i, t_1) \in C$. Since, by hypothesis, x does not exist at time t_1, x is absent from (i, t_1), and is not garbage in C.

Now, let us reason about case 4. We call T the transaction which takes place between times t_1 and $t_1 + 1$. Since x exists at time t_1 but not at time $t_1 + 1$, transaction T deletes x. By rule 6 in Section 2.4, T has therefore access to x. Let us show that the initial access of T to x is obtained according to rule 3 in Section 2.4. *A priori*, this access may be obtained through any of rules 1, 2 or 3. But in fact, rule 1 cannot apply because x does not exist at time $t_1 + 1$, and thus is not root. Rule 2 cannot apply because x can only be created by T once T has access to x: x exists when T begins, and therefore T can create x only after destroying it; the destruction, in turn, is only possible once T has access to x.

Since x is accessed by T through rule 3, T has access to a pointer value p such that $x = *p$. By lemma 4, there exists then an integer $d_p \geq 0$ and objects $x_0^p, ..., x_{d_p}^p$ such that

A1: x_0^p is root;

A2: for every integer $j \in [0 \,..\, d_p - 1]$, at time t_1 object x_j^p exists and contains a pointer to x_{j+1}^p;

A3: at time t_1, object $x_{d_p}^p$ exists and contains a copy of p;

A4: for every integer $j \in [0 \,..\, d_p]$, transaction T locks part $P(x_j^p)$.

We define the integer $e = d_p + 1$, and objects $x_0, ..., x_e$ such that $x_j = x_j^p$ for $j \in [0 \,..\, d_p]$ and $x_e = x$. Then, according to Definition 6, in order to prove that x is reachable in C, it suffices to establish the following statements.

S1: x_0 is a root;

S2: $x_e = x$;

S3: for every integer $j \in [0\,..\,e]$, some snapshot in C contains a copy of x_j which, except for $j = e$, contains a pointer to x_{j+1}.

Statement S1 is equivalent to A1. Statement S2 is one of our hypotheses. Let us establish S3. To prove that the statement holds for any given $j \in [0\,..\,e]$, it is sufficient to prove that the two following facts hold for this j:

S3.1: snapshot $(P(x_j), t_1)$ contains a copy of x_j which, except for $j = e$, points to x_{j+1}.

S3.2: $(P(x_j), t_1) \in C$;

For $j = e$, statement S3.1 directly results from the hypotheses for case 4 (by definition, $x_e = x$); for $j = e - 1$, it results from statement A3; for $j < e - 1$, it results from A2. To prove S3.2, we use path H defined as follows:

$$\begin{aligned} &\text{for } t \leq t_1, \qquad H(t) = P(x_j) \\ &\text{for } t > t_1, \qquad H(t) = i \end{aligned}$$

According to Definition 9, for $j = e$, H is indeed a path because in this case $P(x_j) = i$ and H is a constant function. For $j < e$, to show that H is indeed a path, we just need to prove that T locks $P(x_j)$ for reading or writing, and locks i for writing. The former results from statement A4. The latter results from the fact that x is deleted by T.

Since $(i, t_1 + 1) \in C'$, part i is captured in C at time $t_1 + 1$. Since $H(t_1 + 1) = i$, reasoning as in case 3 we obtain $(H(t_1), t_1) \in C$, *i.e.* $(x_j, t_1) \in C$, and S3.2 holds. This completes the proof of statement S3, and of the fact that x is not garbage in C in case 4. □

Proof of proposition 1 Let us take an arbitrary object x which is not garbage in C', and prove that x is not garbage in C. We are in one of the two following cases.

Case 1: C' contains no copies of x.

Case 2: x is reachable in C'.

In case 1, it suffices to observe that since C' is a cut, a time t such that $(P(x), t) \in C'$ exists. Since C' contains no copies of x, object x does not exist at time t. The hypotheses of proposition 2 are therefore met; by applying the proposition, we deduce that x is not garbage in C. This terminates the proof for case 1.

Consider now case 2. Let d represent the reachability index of x in C'. We reason by induction on d. In order to prove that x is not garbage for $d = 0$, it suffices to observe that by Definition 15, either x is a root, or C' contains a snapshot $(P(x), t)$ in which x is nonexistent. In the first case, x is trivially reachable in C; in the second case, it is not garbage in C by proposition 2.

Now, assume that $d > 0$. There exists then an object x' such that some copy of x' present in C' contains a pointer to x, and that x' is reachable in C' with reachability index $d-1$; the existence of x' results from Definition 15 (easy proof omitted). We define

$$\begin{aligned} i &= P(x) \\ i' &= P(x') \end{aligned}$$

Let t be a time such that snapshot (i', t) belongs to C' and contains a copy of x' which points to x. The induction hypothesis allows us to assume that x' is not garbage in C.

We are in one of the following cases.

Case 1: $t > t_1 + 1$

Case 2: $t = t_1 + 1$ and $(i', t) \in C$

Case 3: $t = t_1 + 1$ and $(i', t) \notin C$ and at time t_1, object x' exists and contains a pointer to x

Case 4: $t = t_1 + 1$ and $(i, t) \notin C$ and at time t_1, object x' does not contain a pointer to x (x' may or may not exist at time t_1)

The rest of the proof is done for each case separately. For cases 1 and 2, we have $(i', t) \in C$. Thus, a copy of x' present in C contains a pointer to x. Since x' is not garbage in C, this implies that x is reachable in C.

In case 3, reasoning as in the proof of Proposition 2, case 3, we obtain $(i', t_1) \in C$. This, together with the hypotheses for case 3 and with the fact that x' is not garbage in C, implies that x is reachable in C.

Now, consider case 4. We call T the transaction which takes place between times t_1 and $t_1 + 1$. Let us consider two subcases. In the first subcase, we assume that x does not exist at time t_1 and T creates x. We consider then path H, defined as follows:

$$\begin{aligned} &\text{for } t \leq t_1, \qquad H(t) = i \\ &\text{for } t > t_1, \qquad H(t) = i' \end{aligned}$$

Since T creates x and writes a pointer into x', T locks for writing parts i and i'. According to Definition 9, H is therefore indeed a path.

Since $(i', t_1 + 1) \in C'$, part i' is captured in C at time $t_1 + 1$. Since, by hypothesis, $(H(t_1 + 1), t_1 + 1) \notin C$, and no snapshot taken before time t_1 belongs to C, we have $(H(t_1), t_1) \in C$, *i.e.* $(i, t_1) \in C$. Given that, by hypothesis, x does not exist at time t_1, C contains therefore a snapshot in which x is nonexistent, and x is not garbage in C.

Consider now the remaining subcase, *i.e.* the situation where x exists at time t_1 or T does not create x. If x is absent from C, then x is not garbage in C. When proving that x is not garbage in C, we can therefore assume without loss of generality that a copy of x is present in C.

Let us call p the pointer value which points to x. According to the hypotheses for this case, T writes p into x', and therefore has access to p. Since p points to a non-root object which exists at time

t_1 or is not created by T, we can use lemma 4, and say that there exists an integer $d_p \geq 0$ and objects $x_0^p, ..., x_{d_p}^p$ which satisfy the conditions quoted under labels A1–A4 in the proof of proposition 2.

The rest of the proof is identical to the part of the proof of proposition 2 which follows conditions A1–A4, with the two following differences. First, in our case the statements S3.1 and S3.2 do not necessarily hold for $j = e$. Therefore, to prove that S3 holds for $j = e$, instead of S3.1–S3.2 we use the assumption made above that C contains a copy of x.

Second, for $j < e$ the proof of statement S3.2 is slightly different: we use index i' instead of i. The modified proof is as follows. We define path H' as follows:

$$\begin{aligned} &\text{for } t \leq t_1, \qquad H'(t) = P(x_j) \\ &\text{for } t > t_1, \qquad H'(t) = i' \end{aligned}$$

To show that H' is indeed a path, we just need to prove that T locks $P(x_j)$ for reading or writing, and locks i' for writing. The former results from statement A4. The latter results from the fact that, by hypothesis, x' is modified by T.

Since $(i', t_1 + 1) \in C'$, part i' is captured in C at time $t_1 + 1$. Since $H'(t_1 + 1) = i'$, we obtain $(H(t_1), t_1) \in C$, *i.e.* $(P(x_j), t_1) \in C$, and S3.2 holds. □

A.5 Proof of Theorem 5 (sum of bank accounts)

Let us first show two lemmas.

Lemma 5 *Let C be a GC-consistent simple cut. Let C' be the temporal restriction after t_0 of C. For every i, we define t be the time such that $(i, t) \in C$. Then,*

1. *if $t > t_0$, then $(i, t) \in C'$ and for any $t' \neq t$, $(i, t') \notin C'$;*
2. *otherwise, $(i, t_0) \in C'$ and for any $t' \neq t_0$, $(i, t') \notin C'$;*

Proof Let C, t_0, C', i and t be defined as in the lemma. Consider the two cases $t \leq t_0$ and $t > t_0$. In the first case, we recall that after t_0, C and C' coincide. This implies that C' contains no snapshots of part i taken after t_0. Since C' contains no snapshots taken before t_0, we deduce that C' may contain only one snapshot of part i, namely (t_0, i). Since, by lemma 3, C' is a cut and therefore contains at least one snapshot of i, we have $(i, t_0) \in C'$. These considerations imply that the lemma holds for those values i for which $t \leq t_0$.

Now, consider the case $t > t_0$. We define the constant path H such that $H(t) = i$ for every t. Path H does not cross C at time t_0 or earlier. This implies that part i is not captured by C at time t_0. Thus, $(i, t_0) \notin C'$. This and the fact that C' contains no snapshots taken before t_0 allow us to say that C' may only contain snapshots of part i taken at times $t' > t_0$. And since C and C' coincide for all such times, one and only one snapshot of i taken after t_0 may exist in C', namely (i, t). □

Lemma 6 *Let T be the transaction which takes place in E between times t_0 and $t_0 + 1$. Let W be the set of indices i such that part i is locked for writing by T. Let C be a GC-consistent simple cut. Then, at least one of the two following statements holds*

1. *for every $i \in W$, the snapshot of i in C is taken at time t_0 or before;*
2. *or for every $i \in W$, the snapshot of i in C is taken at time $t_0 + 1$ or after.*

Proof We prove the lemma by contradiction. Assume on the contrary that, under the lemma's hypotheses, both statements are false. Since C contains one and only one snapshot of every part, this is equivalent to the following two lines:

- for some $i \in W$, we have $(i, t) \in C$ for a value $t \geq t_0 + 1$
- and for some $i' \in W$, we have $(i', t') \in C$ for a value $t' \leq t_0$

Under this assumption, we introduce path H, defined as follows:

$$\begin{aligned} &\text{for } t \leq t_0, \qquad H(t) = i \\ &\text{for } t > t_0, \qquad H(t) = i' \end{aligned}$$

Since between times t_0 and $t_0 + 1$ parts i and i' are both locked for reading and writing, H is indeed a path. H does not cross C (the simple proof of this fact is left out). Therefore, C is not a GC-consistent cut. Thus contradiction. □

Now, let us use lemmas 5 and 6 to establish Theorem 5. The proof is done by an induction similar to the first induction in Section A.4. We assume that E satisfies the hypotheses of Theorem 5 and that C is a GC-consistent simple cut of E. To establish Theorem 5, it is sufficient to prove that relation (1) in the theorem holds under these hypotheses. For this purpose, we introduce notation t_1 and t_2 for the times when, respectively, the first and the last snapshot in C is taken. Our proof is done by induction on the duration of C, which is equal to $t_2 - t_1$. We omit the very simple proof for null durations. For non-null durations, we define C' to be the temporal restriction of C after $t_1 + 1$. Lemmas 3 and 5 imply that C' is a GC-consistent simple cut. Reasoning as in Section A.4, we observe that C' only contains snapshots taken between times $t_1 + 1$ and t_2. Its duration is therefore strictly less than the duration of C. This allows us to apply the induction hypothesis and to assume the following

$$S = \sum_{j=0}^{l} v_j^{C'} \tag{2}$$

Given this relation, all we have to prove is

$$\sum_{j=0}^{l} v_j^{C} = \sum_{j=0}^{l} v_j^{C'} \tag{3}$$

Let us call T the transaction which occurs between times t_1 and $t_1 + 1$. We call W the set of indices which correspond with parts locked by T for writing. We call W' the set of indices $j \in [0\,..\,l]$ such that $P(v_j) \in W$. In other words, W' is the set of indices j such that T has the possibility to

modify v_j. Consider the following statements:

$$\text{for every } j \in [0\,..\,l] \text{ satisfying } j \notin W', \quad v_j^C = v_j^{C'} \tag{4}$$

$$\sum_{j \in W'} v_j^C = \sum_{j \in W'} v_j^{C'} \tag{5}$$

These two statements trivially imply (3). The rest of this proof consists in establishing them. To establish (4), we consider an arbitrary index $j \notin W'$, and we prove that $v_j^C = v_j^{C'}$. Let t be the time such that $(P(v_j),\, t) \in C$. We consider two cases: either $t \geq t_1 + 1$ or $t = t_1$. For $t \geq t_1 + 1$, lemma 5 implies that $(P(v_j),\, t) \in C'$. Accordingly, the values v_j^C and $v_j^{C'}$ both represent the integer v_j^t, and are therefore equal. For $t = t_1$, lemma 5 implies that $(P(v_j),\, t_1 + 1) \in C'$. We have therefore $v_j^C = v_j^{t_1}$ and $v_j^{C'} = v_j^{t_1+1}$. Since $j \notin W'$, transaction T does not modify the part containing v_j, and we have $v_j^{t_1} = v_j^{t_1+1}$. This implies that $v_j^{C'} = v_j^C$, and completes the proof of (4).

Now, let us establish (5). According to lemma 6, we are in one of the two following cases:

1. for every $i \in W$, $(i, t) \in C$ for some $t \geq t_1 + 1$
2. or for every $i \in W$, $(i, t_1) \in C$

In the first case, lemma 5 implies that for every $j \in W'$, simple cuts C and C' contain the same snapshot of $P(v_j)$. This implies that for every $j \in W'$, the values v_j^C and $v_j^{C'}$ represent the same number. This, in turn, implies (5).

In the second case, observe that for $j \notin W'$, variable v_j remains constant between times t_1 and $t_1 + 1$. Therefore, the following equalities hold:

$$\sum_{j \notin W'} v_j^{t_1} = \sum_{j \notin W'} v_j^{t_1+1} \tag{6}$$

$$S - \sum_{j \notin W'} v_j^{t_1} = S - \sum_{j \notin W'} v_j^{t_1+1}$$

$$\sum_{j \in W'} v_j^{t_1} = \sum_{j \in W'} v_j^{t_1+1} \tag{7}$$

To establish equality (5), it is now sufficient to show that each side of (5) is equal to the corresponding side of (7). The left sides are equal because we are in the case in which for every $i \in W$, $(i, t_1) \in C$, and thus for every $j \in W'$, we have $v_j^C = v_j^{t_1}$. For the right hand sides, the proof is analogous, except that lemma 5 is used to state that $v_j^{C'} = v_j^{t_1+1}$ for every $j \in W'$. □

A.6 Proof of Theorem 3 (characterization of causal cuts)

We use the following lemma, which directly results from Definitions 9 and 13.

Lemma 7 *For every i, i', t and t', relation*

$$(i, t) \to (i', t')$$

holds iff $t \leq t'$ and there exists a path H in E such that $H(t) = i$ and $H(t') = i'$.

Theorem 3 says that in any given execution E, a cut C is causal iff it crosses every path H in E in exactly one point. To establish the theorem, we first assume that C crosses every path in exactly one point, and we prove that it is causal, *i.e.* that it satisfies conditions 1 and 2 in Definition 14.

To show that C satisfies condition 1, we proceed by contradiction: we assume on the contrary that there are two different snapshots $(i,t) \in C$ and $(i',t') \in C$ satisfying $(i,t) \to (i',t')$. By lemma 7, there exists then a path H such that $H(t) = i$ and $H(t') = i'$. C crosses H at two points, namely (i,t) and (i',t'). Thus contradiction.

To show that C satisfies condition 2, we observe that if a transaction occurring between times t_0 and $t_0 + 1$ reads part i and writes part i', then a path H can be defined as follows.

for $t \leq t_0$, $H(t) = i$

for $t \geq t_0 + 1$, $H(t) = i'$

Now, let us proceed by contradiction. We assume that condition 2 in Definition 14 does not hold, *i.e.* that conditions 2a and 2b are both false. Then,

no value $t \leq t_0$ satisfies the condition $(i,t) \in C$

and no value $t \geq t_0 + 1$ satisfies the condition $(i',t) \in C$

This and the definition of H imply that we do not have $(H(t),t) \in C$ for any value t. This means that H does not cross C. Thus contradiction. This completes the proof that a cut which crosses every path in exactly one point is causal.

Now, let us show the converse: we assume that C is a causal cut of E, and we prove that it crosses every path in E in exactly one point. First, observe that C does not cross any path in more than one point: by lemma 7, two snapshots belonging to the same path cannot be causally incomparable, and thus cannot both belong to C.

We still need to prove that C crosses every path H. We proceed by induction on the number $v(H)$, defined to be the number of values t such that t and $t+1$ are integer clock values and $H(t) \neq H(t+1)$. For $v(H) = 0$, H is a constant path, and C crosses H because it contains at least one snapshot of every part.

For $v(H) > 0$, we define t_0 to be the largest integer such that t_0 and $t_0 + 1$ are integer clock values, and $H(t_0) \neq H(t_0 + 1)$. We define path H', which satisfies $v(H') = v(H) - 1$, as follows:

for $t \leq t_0$, $H'(t) = H(t)$

for $t \geq t_0 + 1$, $H'(t) = H(t_0)$

(We omit the easy proof that H' is indeed a path and that $v(H') = v(H) - 1$.) The induction hypothesis allows us to say that for some t', we have $(H'(t'),t') \in C$. In the case $t' \leq t_0$, the proof is extremely simple: we have $H(t') = H'(t')$, thus H crosses C at snapshot $(H(t'),t')$.

The proof is more complicated in the remaining case, *i.e.* if $t' \geq t_0 + 1$. First, we recall that $H(t_0 + 1) \neq H(t_0)$. From this fact and from the definition of paths (Definition 9), we deduce that between times t_0 and $t_0 + 1$, a transaction reads part $H(t_0)$ and writes part $H(t_0 + 1)$. Condition 2 in Definition 14 implies therefore that at least one of the following holds.

(i) The snapshot of part $H(t_0)$ in C is taken at or before time t_0,

(ii) or the snapshot of part $H(t_0+1)$ in C is taken at or after time t_0+1.

From the fact that $(H'(t'),t') \in C$ and from equality $H'(t') = H(t_0)$, we can deduce that $(H(t_0),t') \in C$. This and the facts that $t' > t_0$ and that C is a simple cut imply that *(i)* is false. *(ii)* is therefore true, and for some $t'' > t_0$, we have $(H(t_0+1),t'') \in C$. Since $H(t'') = H(t_0+1)$, this means that H crosses C. □

A.7 Proof of Theorem 4 (detection of dangling pointers)

In order to prove the theorem, we introduce the following lemma:

Lemma 8 *Assume that object x is created exactly once in execution E, and the creation occurs in transaction T, which takes place between times t_0 and t_0+1. Assume also that no pointers to x exist at time 0. Let*

$$i_0 = P(x)$$

Then, pointers to x can only be present in those snapshots (i,t) which satisfy $(i_0,t_0) \rightarrow (i,t)$. No pointers to x are present in (i_0,t_0).

Instead of the full proof of the lemma, which is technical and boring, we give an informal description. When x is created, a pointer to x is created by the system and made accessible to T. Then, T can store the pointer in all the pages to which it has write access. All the pointers to x present in the system are obtained by successively copying this pointer (this results from the rules about pointers in Section 2.4, and from the fact that at the beginning, the system contains no pointers to x). A snapshot (i,t) may therefore contain a pointer to x only if there is a path going from (i_0,t_0) to (i,t), *i.e.* if (i_0,t_0) causally precedes (i,t). Snapshot (i_0,t_0) cannot contain pointers to x because it is taken before the execution of T. □

Now, let us consider a causal cut C of E. We assume that no objects are destroyed and no dangling pointers ever exist in E. To prove Theorem 4, it is sufficient to show under these assumptions that C contains no dangling pointers, *i.e.* that for every pointer p present in C, an object pointed to by p is also present in C.

Since no pointer in E is ever dangling and p, being present in C, is also present at some time in E, we know that an object x pointed to by p exists in E at some time. Let us call i_0 the part containing x.

We consider two cases, depending on whether x exists at time 0. If x exists at time 0, the facts that x exists all the time during E (because it cannot be destroyed), and that C contains a snapshot of part i_0 imply that C contains a copy of x. Therefore, p is not dangling in C.

In the remaining case, *i.e.* if x is created after time 0, we define t_0 to be the time such that x is created between times t_0 and t_0+1 (x cannot be created more than once because we assume that no object is destroyed during E). We proceed by contradiction: we assume that a copy of p is present in some snapshot $(i_1,t_1) \in C$, yet x is absent from C. Let t_2 be the time such that $(i_0,t_2) \in C$. Since

x is not present in C and is present in part i_0 from time $t_0 + 1$ to the end of the execution, we know that $t_2 \leq t_0$. This implies the following causal precedence

$$(i_0, t_2) \quad \rightarrow \quad (i_0, t_0) \tag{8}$$

Since no dangling pointers appear in E, the non-existence of x at time 0 implies that no pointer to x exists in E at time 0. We are therefore entitled to use lemma 8, which allows us to deduce the following relations

$$(i_0, t_0) \quad \rightarrow \quad (i_1, t_1) \tag{9}$$
$$(i_0, t_0) \quad \not\rightarrow \quad (i_1, t_1) \tag{10}$$

From relations (8–10) and from the fact that causal precedence is a partial order, we deduce that

$$(i_0, t_2) \quad \rightarrow \quad (i_1, t_1) \tag{11}$$
$$(i_0, t_2) \quad \not\rightarrow \quad (i_1, t_1) \tag{12}$$

Lemma 7 and relations (11–12) imply that C contains two different snapshots belonging to the same path. By Theorem 3, this implies that C is not a causal cut. Thus contradiction. □

References

[1] S. Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In *International Conference on Parallel Processing and Applications*, pages 243–246, L'Aquila, Italy, September 1987. Elsevier North Holland.

[2] Guy T. Almes. Garbage collection in an object-oriented system. Technical Report CMU-CS-80-128, Carnegie Mellon University, June 1980.

[3] Laurent Amsaleg, Michael Franklin, and Olivier Gruber. Efficient incremental garbage collection for client-server object database systems. In *Proceedings of the 21th VLDB International Conference*, Zurich, Switzerland, September 1995.

[4] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *SIGPLAN Notices*, 23(7):11–20, 1988. Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation.

[5] Ozalp Babaoglu and Keith Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In *BROADCAST First Year Report*, volume 1 (Fundamental Concepts) of *LNCS*, St. Malo, France, October 1993. which one ?

[6] Henry G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[7] François Bancilhon, Claude Delobel, and Paris Kannellakis. *Building an Object-Oriented Database: the O_2 Story*. Morgan Kaufmann, 1991.

[8] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[9] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[10] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–83, January 1994.

[11] EXODUS Project Group. EXODUS storage manager architectural overview. Available at `ftp://ftp.cs.wisc.edu/exodus/sm/doc/arch_overview.3.0.ps`, 1993.

[12] EXODUS Project Group. Using the EXODUS storage manager v3.0. Available at `ftp://ftp.cs.wisc.edu/exodus/sm/doc/using3.0.ps`, 1993.

[13] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[14] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. *SIGPLAN Notices*, 27(10):92–109, October 1992. Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications.

[15] Eliot K. Kolodner. Atomic incremental garbage collection and recovery for a large stable heap. Technical Report MIT/LCS/TR-534, Massachusetts Institute of Technology, February 1992.

[16] C. Lamb, G. Landis, J. Orenstein, and D. Weinred. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[17] Friedmann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[18] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 161–174, Asheville, NC (USA), December 1993.

[19] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), September 1995.

[20] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Saint-Malo (France), September 1992. Springer-Verlag.

[21] Voon-Fee Yong, Jeffrey Naughton, and Jie-Bing Yu. Storage reclamation and reorganization in client-server persistent object stores. In *Data Engineering International Conference*, pages 120–133, Houston, Texas, February 1994.

Contents

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399